Algoritmos e Estruturas de Dados I

Marcos Castilho Fabiano Silva Daniel Weingaertner

Versão 0.7.2
Agosto de 2015

AVISO: Este texto ainda está em construção.

Lista de Figuras

Sumário

Capítulo 1

Introdução

Este material contém notas de aulas da disciplina CI055 – Algoritmos e Estruturas de Dados I, ministrada para os curso de Bacharelado em Ciência da Computação (BCC), Matemática Industrial (MatInd) e para o Bacharelado em Informática Biomédica (BIB) da Universidade Federal do Paraná.

Esta disciplina é ministrada no primeiro semestre (para os calouros) e é a primeira das quatro que cobrem o conteúdo básico de algoritmos sem o qual um curso de computação não faz sentido. As disciplinas subsequentes são:

- Algoritmos e Estruturas de Dados II;
- Algoritmos e Estruturas de Dados III; e
- Algoritmos e Teoria dos Grafos.

A orientação da Coordenação dos cursos é que este deve ter um conteúdo forte em conceitos de algoritmos no qual a implementação final em uma linguagem de programação é vista apenas como um mecanismo facilitador ao aprendizado dos conceitos teóricos.

O texto está dividido em duas partes bem definidas, a primeira entre os capítulos 2 e 9, contém os princípios básicos da construção de algoritmos elementares, incluindo a parte de subprogramas, com especial atenção a questões tais como passagem de parâmetros e variáveis locais e globais. A noção de modularidade é bastante explorada.

A segunda parte, a partir do capítulo 10, contém princípios de estruturas de dados básicas, onde se introduz a noção de vetores unidimensionais, a primeira das estruturas de dados. Nesta estrutura, praticamos a elaboração de algoritmos modulares e já não escrevemos um código inteiro, mas sim um conjunto de funções e procedimentos que, em uma abordagem por refinamentos sucessivos (*top down*), são construídos passo a passo.

Alguns algoritmos importantes são estudados em suas versões básicas: busca e ordenação de vetores. Noções de complexidade de algoritmos são mencionadas, ainda que de modo informal, pois isto é conteúdo de períodos mais avançados. Contudo, é importante ao aprendiz ter noção clara da diferença de custo entre diferentes algoritmos.

As últimas estruturas de dados relevantes para o primeiro período são as matrizes e o conceito de registro. Havendo tempo, também se discute estruturas um pouco mais sofisticadas, misturando-se vetores, registros e matrizes.

Finalmente se oferece um desafio aos alunos. O objetivo é o de mostrar uma aplicação interessante dos conceitos que eles já dominam. Normalmente trabalha-se em sala de aula o desenvolvimento de um programa que tem sido a construção de um jogo simples que pode ser implementado em uma estrutura de matriz, eventualmente com registros. A ideia é que eles possam fazer um programa mais extenso para treinarem a construção de programas modulares. Este material constitui o último capítulo deste material.

O estudante não deve iniciar uma parte sem antes ter compreendido o conteúdo das anteriores. Também não deve iniciar um novo capítulo sem ter compreendido os anteriores.

Sobre a linguagem, o estudante é encorajado a buscar apoio na literatura e nos guias de referência disponíveis para o compilador escolhido (Free *Pascal*), incluindo um guia de referência básico que foi escrito pelos monitores da disciplina no ano de 2009.

A leitura destas notas de aula não isenta o estudante de buscar literatura complementar, sempre bem-vinda. Em particular, uma ótima história da computação pode ser encontrada em [Tre83]. Alguns excelentes textos introdutórios em algoritmos estão em [Car82], [Sal98], [Med05] e [Wir78]. Para mais detalhes de programação em *Pascal* o leitor pode consultar [Far99] e também os guias de referência da linguagem [Gui]. Finalmente, embora talvez de difícil compreensão para um iniciante, recomendamos pelo menos folhear o material em [Knu68].

Capítulo 2

Sobre problemas e soluções

Vamos iniciar nosso estudo com uma breve discussão sobre problemas e soluções. O objetivo é deixar claro desde o início que:

- não existe, em geral, uma única solução para o mesmo problema;
- algumas soluções são melhores do que outras, sob algum critério;
- alguns problemas são casos particulares de outros similares;
- as vezes, é melhor resolver o problema mais genérico, assim, resolve-se uma classe de problemas, e não apenas um.

Serão apresentados dois problemas reais e para cada um deles segue uma discussão sobre a existência de diversas soluções para um dado problema. A ênfase será dada nas diferenças entre as soluções e também na discussão sobre até que ponto deve-se ficar satisfeito com a primeira solução obtida ou se ela pode ser generalizada para problemas similares.

2.1 Contando o número de presentes em um evento

No primeiro dia letivo do primeiro semestre de 2009, um dos autores deste material colocou o seguinte problema aos novos alunos: queríamos saber quantos estudantes estavam presentes na sala de aula naquele momento. A sala tinha capacidade aproximada de 100 lugares e a naquele momento estava razoavelmente cheia.

Os estudantes discutiram várias possibilidades. Apresentamos todas elas a seguir.

Primeira solução

A primeira solução parecia tão óbvia que levou algum tempo até algum aluno verbalizar: o professor conta os alunos um por um, tomando o cuidado de não contar alguém duas vezes e também de não esquecer de contar alguém.

Quais são as vantagens deste método? Trata-se de uma solução simples, fácil de executar e produz o resultado correto. É uma solução perfeita para salas de aula com

poucos alunos, digamos, 20 ou 30. Outro aspecto considerado foi o fato de que este método não exige nenhum conhecimento prévio de quem vai executar a operação, a não ser saber contar. Também não exige nenhum equipamento adicional.

Quais as desvantagens? Se o número de alunos na sala for grande, o tempo necessário para o término da operação pode ser insatisfatório. Para piorar, quanto maior o número, maior a chance de aparecerem erros na contagem. Foi discutida a adequação desta solução para se contar os presentes em um comício ou manifestação popular numa praça pública. Concluiu-se pela inviabilidade do método nestes casos.

Executamos a contagem em aproximadamente 1 minuto. Dois alunos também fizeram a contagem e, após conferência, obtivemos o resultado correto, que serviu para análise das outras soluções.

Segunda solução

Pensando no problema de se contar na ordem de 100 alunos, um estudante sugeriu que se fizesse apenas a contagem das carteiras vazias e em seguida uma subtração com relação ao número total de carteiras na sala.

A solução também é muito simples e funciona perfeitamente bem, mas exige um conhecimento prévio: deve-se saber antecipadamente o total de carteiras na sala.

Esta maneira de contar é cada vez melhor quanto maior for o número de presentes, pois o número de carteiras vazias é menor do que o das ocupadas. Por outro lado, se a sala estiver com pouca gente, o método anterior é mais eficiente.

Os alunos observaram também que a solução não se aplica para os casos de contagem de presentes a um comício numa praça pública, pois não há carteiras na rua.

Terceira solução

Para resolver o problema do comício, outro estudante sugeriu que se fizesse uma estimativa baseada na metragem total da praça, multiplicada pelo número estimado de pessoas por metro quadrado.

Solução elegante, na prática é o que a organização do comício e a polícia usam. Mas deve-se saber de antemão a metragem da praça e estimar a taxa de pessoas por metro quadrado. O método é tão bom quanto melhor for a estimativa. Também é melhor se a população estiver uniformemente distribuída.

Concluiu-se que é um bom método, mas que não é preciso. Isto é, a chance do número estimado ser exatamente o número de presentes é baixa. Os métodos anteriores são exatos, isto é, nos dão o número correto de presentes. Este método também serve razoavelmente bem para o número de alunos na sala de aula. De fato, nesta aula, o professor conseguiu o número com aproximação 80% correta. A questão que restou é se o erro de 20% é aceitável ou não. Isto depende do motivo pelo qual se quer contar os alunos na sala.

Quarta solução

Para resolver o problema da precisão, outro estudante sugeriu o uso de roletas.

Efetivamente é esta a solução para contar torcedores no estádio ou presentes em um show de rock. Mas também foi considerado que a solução exige uma ou mais catracas, uma barreira para ninguém entrar sem passar pela roleta e etc, para se garantir a exatidão do resultado. No caso do comício não seria viável. No caso da sala de aula foi constatado que não havia roletas e portanto o método não se aplicava.

Quinta solução

Mais uma vez outro estudante apresentou uma boa alternativa: contar o número de filas de carteiras e, dado que todas tenham o mesmo número de estudantes, então bastaria uma simples multiplicação para a determinação do número correto.

De fato esta solução funciona perfeitamente bem em lugares como por exemplo o exército. As filas são rapidamente arrumadas com, digamos, 10 soldados em cada fila, sabendo-se o número de filas basta multiplicar por 10, eventualmente tendo-se que contar o número de pessoas em uma fila que não tenha completado 10.

Infelizmente as carteiras estavam bagunçadas na nossa sala e este cálculo não pode ser feito. Também ficaria estranho o professor colocar todos os alunos em filas. Foi também observado que o método fornece a solução exata para o problema.

Sexta solução

Nova sugestão de outro aluno: cada estudante no início de cada fila conta o número de alunos da sua fila, tomando o cuidado de contar a si próprio também. Depois soma-se todas as contagens de todos os primeiros de fila.

Solução muito boa. Na verdade é a versão *em paralelo* da primeira solução. Distribuindo-se a tarefa, cada primeiro de fila tem entre 10 e 15 alunos para contar em sua fila. Se a soma foi correta o número obtido ao final do processo é exato. No caso daquela aula os estudantes realizaram a operação em poucos segundos, mais algum tempo para as somas (isto demorou mais...). Mas o resultado foi exato.

A solução não exige conhecimento prévio, não exige equipamento adicional e é razoavelmente escalável, isto é, funciona para salas de tamanhos diferentes.

Sétima solução

Para finalizar, o professor apresentou a solução seguinte: todos os estudantes se levantam e se atribuem o número 1. Em seguida os alunos se organizam em pares. Em cada par, primeiro é somado o número de cada um dos dois, um deles guarda este número e permanece de pé, o outro deve se sentar. Os que ficaram em pé repetem este processo até que só exista um único aluno em pé. Ele tem o número exato de estudantes na sala.

Como se divide a sala em pares, após a primeira rodada metade da sala deve ter o número 2 e a outra metade está sentada, considerando que a sala tem o número de alunos par. Se for ímpar um deles terá ainda o número 1. Após a segunda rodada um quarto dos alunos deverá ter o número 4 e três quartos estarão sentados, eventualmente um deles terá um número ímpar. É fácil perceber que o resultado sai em tempo

proporcional ao logaritmo do número total de alunos, o que é bem rápido. De fato, para mil pessoas o processo termina em 10 passos e para um milhão de pessoas termina em 20 passos.

Parece um bom algoritmo, ele dá resultado exato, não exige conhecimento prévio, é escalável, isto é, funciona muito bem para um grande número de pessoas, mas exige organização dos presentes.

Infelizmente aquela turma não se organizou direito e o resultado veio com um erro de 40%... Mas após duas rodadas de treinamento, na terceira conseguimos obter o resultado correto.

2.2 Trocando os quatro pneus

Todo mundo sabe trocar pneus, embora não goste. O processo que um cidadão comum executa é muito simples: levanta o carro com o macaco, tira todos os quatro parafusos da roda com o pneu furado, tira a roda do eixo, coloca a roda com o pneu novo no eixo, em seguida aperta os quatro parafusos. Finalmente, baixa o carro e está pronto.

Nos anos 1980, um famoso piloto de fórmula 1 imaginou que poderia ser campeão do mundo se pudesse usar um composto de pneu mais mole e com isto ganhar preciosos segundos com relação aos seus concorrentes. O problema é que estes compostos mais moles se deterioram rapidamente exigindo a troca dos quatro pneus no meio da corrida. O tal piloto, após alguns cálculos, concluiu que se levasse menos de 8 segundos para trocar os quatro pneus, valeria a pena aplicar este método.

Obviamente a solução caseira não serve. O método descrito acima custa em geral 20 minutos por pneu, com um pouco de prática 10 minutos. Com muita prática 2 ou 3 minutos. Para trocar os quatro pneus, 8 a 12 minutos.

Daí o problema: Como trocar os quatro pneus de um carro em menos de 8 segundos?

Um dos grandes custos de tempo é ter que trocar o macaco para cada roda: usamos um macaco hidráulico, destes de loja de pneus, e levantamos o carro todo de uma só vez.

Mas, para cada roda, temos 4 parafusos, isto é, 16 no total, ou melhor, 32, pois tem que tirar e depois recolocar: usa-se uma aparafusadeira elétrica para amenizar o problema, mas ainda não é suficiente.

Se a roda tiver um único parafuso a economia de tempo é maior ainda. Mas ainda estamos na casa dos minutos, e o tempo total deve ser menor que 8 segundos. Desistimos do campeonato?

Com 4 pessoas, cada uma troca uma roda, divide-se o tempo por 4. Opa! Já estamos abaixo de 1 minuto.

Se tiver ainda a possibilidade de 3 pessoas por roda: um tira o parafuso, outro tira a roda velha, um terceiro coloca a roda nova e o primeiro aperta o parafuso. Mais 2 mecânicos para levantar e baixar o carro todo de uma vez e está feito.

Hoje em dia se trocam os quatro pneus de um carro de fórmula 1, com direito a completar o tanque de gasolina, em cerca de 6 segundos.

Ah, o tal piloto foi campeão naquele ano, pois também usou o truque de aquecer

os pneus antes da prova e andar com o carro contendo metade da gasolina, já que ele ia ter que parar nos boxes de qualquer maneira para trocar os pneus... O cara é um gênio.

2.3 Conclusão

Mesmo para um problema simples existem diversas soluções. A escolha da melhor depende de vários fatores. Por exemplo, se a resposta deve ser exata ou não ou se os conhecimentos prévios necessários estão disponíveis, e assim por diante.

É importante notar que somente após uma série de considerações é possível escolher a melhor técnica e somente em seguida executar a tarefa.

Algumas soluções existem a noção de *paralelismo*. Hoje em dia os computadores vêm com vários núcleos de processamento e sempre existe a chance de se tentar quebrar um problema em vários outros menores e deixar que vários processadores resolvam seus pedaços de solução e depois tentar juntar os resultados com mais alguma operação simples.

No caso da fórmula 1 isto funcionou, mas em geral não é verdade. Infelizmente existe o problema da dependência de dados. Por exemplo, o mecânico que vai colocar a roda nova só pode trabalhar depois que o outro tirou a roda velha. Em problemas com alto grau de dependência, paralelizar é complicado.[1].

[1]Não se estudarão algoritmos paralelos nesta disciplina.

Capítulo 3

Sobre algoritmos e programas

Após o estudo do problema, análise das diversas possibilidades de solução e a escolha da melhor delas, cabe agora a tarefa de escrever um programa que implemente esta solução. Antes, contudo, é preciso saber a diferença entre um algoritmo em um programa. Isto será discutido neste capítulo.

3.1 O que é um algoritmo?

Um algoritmo é uma sequência extremamente precisa de instruções que, quando lida e executada por uma outra pessoa, produz o resultado esperado, isto é, a solução de um problema. Esta sequência de instruções é nada mais nada menos que um registro escrito da sequência de passos necessários que devem ser executados para manipular informações, ou dados, para se chegar na resposta do problema.

Isto serve por dois motivos: o primeiro é que através do registro se garante que não haverá necessidade de se redescobrir a solução quando muito tempo tiver passado e todos tiverem esquecido do problema; o outro motivo é que, as vezes, queremos que outra pessoa execute a solução, mas através de instruções precisas, de maneira que não haja erros durante o processo. Queremos um *algoritmo* para a solução do problema.

Uma receita de bolo de chocolate é um bom exemplo de um algoritmo (a lista de ingredientes e as quantidades foram omitidas, bem como a receita da cobertura):

```
Bata em uma batedeira a manteiga e o açúcar. Junte as gemas uma a uma
até obter um creme homogêneo. Adicione o leite aos poucos. Desligue a
batedeira e adicione a farinha de trigo, o chocolate em pó, o fermento
e reserve. Bata as claras em neve e junte-as à massa de chocolate
misturando delicadamente. Unte uma forma retangular pequena
com manteiga e farinha e leve para assar em forno médio pré-aquecido
por aproximadamente 30 minutos. Desenforme o bolo ainda
quente e reserve.
```

Este é um bom exemplo de algoritmo pois podemos extrair características bastante interessantes do texto. Em primeiro lugar, a pessoa que escreveu a receita não é

necessariamente a mesma pessoa que vai fazer o bolo. Logo, podemos estabelecer, sem prejuízo, que foi escrita por um mas será executada por outro.

Outras características interessantes que estão implícitas são as seguintes:

- as frases são instruções no modo imperativo: *bata isso*, *unte aquilo*. São ordens, não sugestões. Quem segue uma receita *obedece* quem a escreveu;
- as instruções estão na forma sequencial: apenas uma pessoa executa. Não existem ações simultâneas.
- existe uma ordem para se executar as instruções: *primeiro* bata a manteiga e o açucar; *depois* junte as gemas, uma a uma, até acabar os ovos; *em seguida* adicione o leite.
- algumas instruções não são executadas imediatamente, é preciso entrar em um modo de repetição de um conjunto de outras instruções: enquanto houver ovos não usados, junte mais uma gema. Só pare quando tiver usado todos os ovos.
- algumas outras instruções não foram mencionadas, mas são obviamente necessárias que ocorram: é preciso separar as gemas das claras *antes* de começar a tarefa de se fazer o bolo, assim como é preciso *ainda antes* quebrar os ovos.
- algumas instruções, ou conjunto de instruções, podem ter a ordem invertida: pode-se fazer primeiro a massa e depois a cobertura, ou vice-e-versa. Mas nunca se pode colocar no forno a assadeira antes de se chegar ao término do preparo da massa.

Mesmo levando estas coisas em consideração, qualquer ser humano bem treinado em cozinha conseguiria fazer um bolo de chocolate razoável com as instruções acima, pois todas as receitas seguem o mesmo padrão. As convenções que estão implícitas no algoritmo são conhecidas de qualquer cozinheiro, pois seguem um formato padrão.

O formato padrão para algoritmos que vamos considerar é o seguinte:

- as instruções serão escritas uma em cada linha;
- as instruções serão executadas uma a uma, da primeira até a última linha, nesta ordem, a menos que o próprio algoritmo tenha instruções que alterem este comportamento;
- em cada linha, uma instrução faz somente uma coisa;
- tudo o que está implícito deverá ser explicitado.

A figura 3.1 ilustra a receita de bolo de chocolate escrita dentro deste formato padrão.

```
Algoritmo para fazer um bolo de chocolate.
início
     Providencie todos os ingredientes da receita.
     Providencie uma forma pequena.
     Ligue o forno em temperatura média.
     Coloque a menteiga na batedeira.
     Coloque o açucar na batedeira.
     Ligue a batedeira.
     Enquanto um creme homogêneo não for obtido, junte mais uma gema.
     Adicione aos poucos o leite.
     Desligue a batedeira.
     Adicione a farinha de trigo.
     Adicione o chocolate em pó.
     Adicione o fermento.
     Reserve a massa obtida em um lugar temporário.
     Execute o algoritmo para obter as claras em neve.
     Junte as claras em neve à massa de chocolate que estava reservada.
     Misture esta massa delicadamente.
     Execute o algoritmo para untar a forma com manteiga e farinha.
     Coloque a forma no forno.
     Espere 30 minutos.
     Tire a forma do forno.
     Desenforme o bolo ainda quente.
     Separe o bolo em um lugar temporário.
     Faça a cobertura segundo o algoritmo de fazer cobertura.
     Coloque a cobertura no bolo.
fim.
```

Figura 3.1: Algoritmo para fazer bolo de chocolate.

Infelizmente, nem todos conseguem fazer o bolo, pois existem instruções que somente os iniciados decifram:

- "adicione aos poucos";
- "misturando delicadamente";
- "quando o creme fica homogêneo?"...

No caso do computador a situação é pior ainda, pois trata-se de um circuito eletrônico, de uma máquina. Por este motivo, as instruções devem ser precisas e organizadas.

Um algoritmo feito para um computador executar deve tornar explícito todas as informações implícitas. Também deve evitar o uso de frases ambíguas ou imprecisas e deve ser o mais detalhado possível. Também não pode ter frases de significado desconhecido.

Na próxima seção vamos desenvolver melhor este tema.

3.2 O que é um programa?

Um programa é a codificação em alguma linguagem formal que garanta que os passos do algoritmo sejam executados da maneira como se espera por quem executa as instruções.

Vamos imaginar, a título de ilustração, que é a primeira vez que a pessoa entra na cozinha em toda a sua vida e resolve fazer um bolo de chocolate seguindo o algorimo 3.1

O algoritmo 3.1 foi escrito por um cozinheiro para ser executado por um outro cozinheiro, o que não é o caso, pois a pessoa é inexperiente em cozinha e não sabe o que significa "bater as claras em neve". Significa que o novato vai ficar sem o bolo.

O novato precisaria de algo mais detalhado, isto é, de instruções meticulosas de como se obtém claras em neve. Poderia ser algo como ilustrado na figura 3.2.

```
Algoritmo para fazer claras em neve
início
     Repita os três seguintes passos:
        Pegue um ovo.
        Quebre o ovo.
        Separe a clara da gema.
        Coloque somente a clara em um prato fundo.
     Até que todos os ovos tenham sido utilizados.
     Pegue um garfo.
     Mergulhe a ponta do garfo no prato.
     Repita os seguinteis passos:
          Bata a clara com o garfo por um tempo.
          Levante o garfo.
          Observe se a espuma produzida fica presa no garfo
     Quando a espuma ficar presa no garfo, pare.
     Neste ponto suas claras em neve estão prontas.
fim.
```

Figura 3.2: Algoritmo para fazer claras em neve.

Já temos algo mais detalhado, mas nosso inexperiente cozinheiro pode ainda ter problemas: como se separa a clara da gema? Este tipo de situação parece não ter fim. Qual é o limite do processo de detalhamento da solução?

O problema é que o cozinheiro que escreveu a receita original não sabia o nível de instrução de quem ia efetivamente fazer o bolo. Para isto, é preciso que se estabeleça o nível mínimo de conhecimento para quem vai executar, assim quem escreve sabe até onde deve ir o nível de detalhamento de sua receita.

Um programa, neste sentido, é um algoritmo escrito de forma tão detalhada quanto for necessário para quem executa as instruções. O algoritmo pode ser mais genérico, o programa não.

Como estamos pensando em deixar que o computador execute um algoritmo, precisamos escrever um programa em uma linguagem que o computador possa entender

as instruções para posteriormente poder executá-las com sucesso.

Qual é, afinal, o conjunto de instruções que o computador conhece? Para responder a esta pergunta precisamos conhecer melhor como funciona um computador, para, em seguida, continuarmos no estudo de algoritmos.

3.3 Exercícios

1. Escreva algoritmos como os que foram escritos neste capítulo para cada uma das soluções do problema discutido na seção 2.1.

2. Escreva um algoritmo para o problema da troca de um único pneu de um carro.

3. Escreva um algoritmo para o problema de trocar um pneu de uma bicicleta.

Capítulo 4

O modelo do computador

Esta seção tem dois objetivos, o primeiro é mostrar como é o funcionamento dos computadores modernos, isto é, no nível de máquina. A segunda é que o aluno perceba, desde o início do seu aprendizado, as limitações a que está sujeito quando programa, e quais são todas as instruções que o computador conhece.

Ao final da leitura, o estudante deve compreender que, por mais sofisticada que seja a linguagem de programação utilizada, a computação de verdade ocorre como será mostrado aqui.[1]

4.1 Histórico

Um computador (hardware) é um conjunto de circuitos eletrônicos que manipulam sinais elétricos e que são capazes de transformar sinais de entrada em sinais de saída. Os sinais elétricos podem ser representados, basicamente, pelos números zero e um. Existem várias maneiras de se fazer isto, mas não entraremos em detalhes neste texto.

O importante a destacar é que uma computação é uma manipulação de dados residentes em memória através de alterações de sinais elétricos realizadas por circuitos integrados implementados normalmente em placas de silício.

Quando os computadores foram criados, na década de 1930, a programação deles era feita de maneira muito precária. Era necessário configurar uma situação dos circuitos para manipular os sinais elétricos da maneira desejada para cada programa particular. Para se executar outro programa era necessário alterar os circuitos, assim se reprogramando o computador para manipular os dados de outra maneira.

Um computador era algo raro naqueles tempos, e devia rodar vários programas diferentes, o que resultava em imenso trabalho para os engenheiros (os programadores eram engenheiros na época).

A memória do computador, naqueles tempos, era exclusividade dos dados que seriam manipulados. O programa era feito nos circuitos eletrônicos.

[1] O texto que segue foi adaptado de outro escrito pelo prof. Renato Carmo para a disciplina CI-208 - Programação de Computadores ministrada para diversos cursos na UFPR.

John von Neumann propôs um modelo bastante simples, no qual tanto o programa quanto os dados poderiam ficar simultaneamente em memória, desde que a parte que ficaria programada nos circuitos pudesse interpretar o que era dado e o que era o programa e realizar os cálculos, isto é, manipular os dados.

Isto foi possível pela implementação em hardware de um limitado conjunto de instruções que são usados pelo programa que está em memória. Isto revolucionou a arte da programação. Os computadores modernos ainda funcionam assim.

Nesta seção pretende-se mostrar através de um exemplo os princípios deste modelo.

4.2 Princípios do modelo

Conforme explicado, Von Neumann propôs que os dados e o programa poderiam ser carregados em memória ao mesmo tempo. Um elemento adicional denominado *ciclo de execução de instruções* controla a execução do programa.

A ideia é implementar em hardware um pequeno conjunto de instruções que não mudam e programar o computador para realizar operações complexas a partir da execução de várias instruções básicas da máquina.

Cada fabricante define o seu conjunto de instruções básicas, mas o importante a observar é que, uma vez implementadas, este conjunto define *tudo o que o computador sabe fazer*. É isto que queremos saber.

Neste capítulo vamos usar como exemplo um computador fabricado pela *Big Computer Company* (BCC).

4.2.1 Endereços versus conteúdos

O computador da BCC implementa o modelo Von Neumann, logo, sua memória contém os dados e o programa.

A memória do computador em um dado instante do tempo é uma configuração de sinais elétricos que podem ser vistos pelo ser humano como uma sequência absurda de zeros e uns (chamados de *bits*).[2]

O ser humano costuma não gostar muito desta forma de visualização, então convencionou algumas maneiras de enxergar números inteiros que representam os bits. Não vamos apresentar neste texto as diversas maneiras de conversão de números, o leitor interessado pode estudar sobre *representação binária* na literatura.

Vamos imaginar que a memória do computador é uma tabela contendo índices (endereços) com conteúdos (dados). A título de exemplo, vamos considerar uma "fotografia" da memória de um computador da BCC em um certo momento, fotografia esta apresentada na figura 4.1

[2]Quem assistiu ao filme Matrix pode imaginar a complicação.

Endereço	Conteúdo
0	1
1	54
2	2
3	1
4	50
5	4
6	7
7	46
8	4
9	47
10	46
11	46
12	7
13	48
14	4
15	49
16	50
17	48
18	3
19	51
20	47

Endereço	Conteúdo
21	49
22	6
23	52
24	51
25	3
26	53
27	46
28	52
29	5
30	55
31	53
32	54
33	8
34	55
35	2
36	56
37	46
38	52
39	5
40	57
41	56

Endereço	Conteúdo
42	54
43	8
44	57
45	9
46	33
47	2
48	76
49	67
50	76
51	124
52	14
53	47
54	235
55	35
56	23
57	78
58	243
59	27
60	88
61	12
62	12

Figura 4.1: Uma fotografia da memória.

Para melhor entendimento, é importante que o leitor tenha em mente a diferença entre *endereço* e *conteúdo do endereço*. Para facilitar a compreensão, vamos adotar uma notação. Seja p um endereço. Denotamos por $[p]$ o conteúdo do endereço p. Vejamos alguns exemplos com base na figura 4.1:

$$[0] = 1$$
$$[[0]] = [1] = 54$$
$$[[[0]]] = [[1]] = [54] = 235$$
$$[0] + 1 = 1 + 1 = 2$$
$$[0 + 1] = [1] = 54$$
$$[[0] + 1] = [1 + 1] = [2] = 2$$
$$[[0] + [1]] = [1 + 54] = [55] = 35$$

4.2.2 O repertório de instruções

Conforme mencionado, o modelo Von Neumann pressupõe que o computador que está em uso possui um conjunto limitado de instruções programado em hardware.

Cada equipamento tem o seu repertório de instruções. O repertório do computador da BCC foi definido após longas discussões da equipe técnica da empresa e tem um conjunto extremamente limitado de instruções, na verdade apenas nove.

Esta definição levou em conta a capacidade financeira da empresa que não tinha muita verba para gravar em circuitos integrados um conjunto mais rico de instruções.

As nove instruções do computador da BCC são apresentadas na figura 4.2.[3].

Código	**Mnemônico**	**Descrição**	**Notação**
1	`load`	escreva em $[p+1]$ o valor do número em $[p+2]$ e some 3 em p	$[p+1] \leftarrow [p+2]$.
2	`add`	escreva em $[p+1]$ o valor da soma dos números em $[[p+2]]$ e $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] + [[p+3]]$.
3	`sub`	escreva em $[p+1]$ o valor da subtração do número em $[[p+2]]$ pelo número em $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] - [[p+3]]$.
4	`mult`	escreva em $[p+1]$ o valor do produto dos números em $[[p+2]]$ e $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] \times [[p+3]]$.
5	`div`	escreva em $[p+1]$ o valor da divisão do número em $[[p+2]]$ pelo número em $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow \frac{[[p+2]]}{[[p+3]]}$.
6	`sqrt`	escreva em $[p+1]$ o valor da raiz quadrada de $[[p+2]]$ e some 3 em p	$[p+1] \leftarrow \sqrt{[[p+2]]}$.
7	`read`	leia um número do teclado, escreva-o em $[p+1]$ e some 2 em p	$[p+1] \leftarrow \underline{\vee}$.
8	`write`	escreva $[[p+1]]$ na tala e some 2 em p	$\square \leftarrow [[p+1]]$.
9	`stop`	pare	$\bullet$

Figura 4.2: O repertório de instruções do computador da BCC.

[3]Os concorrentes comerciais famosos da BCC implementam algumas centenas de instruções, e ainda nenhuma delas é a de bater claras em neve.

4.2.3 O ciclo de execução de instruções

O *ciclo de execução de instruções* define o comportamento do computador. Funciona assim (no computador da BCC):

1. comece com $p = 0$;
2. interprete $[p]$ de acordo com a tabela de instruções e pare somente quando a instrução for uma ordem de parar (instrução 9, `stop`).

Devemos lembrar que este comportamento também está implementado nos circuitos eletrônicos do computador da BCC.

4.2.4 Exemplo de execução de um programa

A grande surpresa por trás do modelo de Von Neumann é que, mesmo que o leitor ainda não compreenda, o que existe na verdade "disfarçado" na fotografia da memória da figura 4.1 é um programa que pode ser executado pelo computador, desde que todo o processo siga as instruções descritas na seção anterior.

Vamos tentar acompanhar passo a passo como é o funcionamento deste esquema. Para isto, o leitor talvez queira ir marcando, a lápis, as alterações que serão feitas a seguir em uma cópia da "fotografia da memória" acima. É recomendado neste momento se ter uma versão impressa daquela página.

Notem que, no momento, não é necessário sabermos qual o programa implementado, afinal de contas, o computador jamais saberá... Ele executa cegamente as instruções. Nós saberemos logo à frente, mas, agora, para entendermos como é o funcionamento deste modelo, vamos nos imaginar fazendo o papel do computador.

1. O programa começa com $p = 0$
2. Em seguida, é preciso interpretar $[p]$, isto é $[0] = 1$. A instrução de código "1" é "`load`", cujo comportamento é, segundo a tabela de instruções "escreva em [2] o valor do número em [3] e some 3 em p". Ora, $[2] = 54$ e $[3] = 2$. Logo, o valor 2 é colocado como sendo o conteúdo da posição 54. Havia nesta posição o valor 235. Após a execução da instrução, existe um 2 neste lugar. O valor 235 não existe mais. Ao final foi somado 3 no valor de p, isto é, agora $p = 3$.
3. Como $p = 3$ devemos interpretar $[3] = 1$. Logo, a instrução é novamente "`load`". Analogamente ao que foi feito no parágrafo anterior, o conteúdo de $[5] = 4$ é colocado como sendo o conteúdo da posição $[4] = 50$. Na posição 50 havia o valor 76. Após a execução da instrução o 76 dá lugar ao 4. Ao final o valor de p foi atualizado para 6.
4. Como $p = 6$ devemos interpretar $[6] = 7$. Logo, a instrução para ser executada agora é "`read`", isto é, esperar o usuário digitar algo no teclado e carregar este valor em $[p + 1] = [7] = 46$. Supondo que o usuário digitou o valor 5, este agora substitui o antigo valor, que era 33. Ao final, o valor de p foi atualizado de 6 para 8.

5. Como $p = 8$ devemos interpretar $[8] = 4$. Logo, a instrução a ser executada é "`mult`". Isto faz com que o computador faça a multiplicação do valor em $[[10]] = [46]$ pelo mesmo valor em $[[11]] = [46]$. O valor em $[46]$ é 5 (aquele número que o usuário tinha digitado no teclado). O resultado da multiplicação, $5 \times 5 = 25$, é carregado na posição de memória $[9] = 47$. O valor ali que era 2 agora passa a ser 25. Ao final, ao valor de p foi somado 4, logo neste momento $p = 12$.

É importante salientar que este é um processo repetitivo que só terminará quando a instrução "`stop`" for a da vez. O leitor é encorajado a acompanhar a execução passo a passo até o final para entender como é exatamente o comportamento dos computadores quando executam programas. Isto é, fica como exercício ao leitor! Destacamos que os circuitos implementados cuidam da alteração do estado elétrico dos circuitos da memória.

4.3 Humanos versus computadores

Nós seres humanos não nos sentimos muito à vontade com este tipo de trabalho repetitivo. Temos a tendência a identificar "meta-regras" e executar a operação com base em um comportamento de mais alto nível. Em suma, nós aprendemos algo neste processo, coisa que o computador só faz em filmes de ficção científica.

A primeira coisa que nos perguntamos é: por qual motivo ora se soma um valor em p, ora outro? Isto é, quando executamos a operação "`load`", o valor somado em p foi 3. Depois, quando executada a operação "`read`" o valor somado foi 2. Em seguida, para a instrução "`mult`" o valor somado foi 4.

O estudante atento, notadamente aquele que foi até o final na execução do ciclo de operações deixado como exercício, talvez tenha percebido uma sutileza por trás deste modelo.

De fato, quando se executa a instrução $[p]$, o conteúdo de $[p+1]$ sempre é o endereço de destino dos dados que são resultado da operação em execução. Os endereços subsequentes apontam para os operandos da operação que está programada para acontecer.

Assim:

- Se for uma multiplicação, subtração ou soma, precisamos de dois operandos e do endereço destino, por isto se soma 4 em p;
- Se for um "`load`" precisamos de um operando apenas, assim como para a raiz quadrada, por isto se soma 3 em p;
- Se for uma leitura do teclado ou uma escrita na tela do computador, então um único argumento é necessário, por isto se soma apenas 2 em p.

Uma outra observação importante é que, por questões de hardware, o computador precisa entender a memória como esta espécie de "tripa". O ser humano, ao contrário,

uma vez que já identificou pequenos blocos relacionados às instruções, pode tentar entender esta mesma memória em outro formato, isto é, separando cada um destes pequenos blocos em uma única linha.

Observamos que apenas os números de 1 a 9 podem ser interpretados como códigos de alguma instrução, pois são os únicos códigos da tabela de instruções da BCC.

A separação dos pequenos blocos resulta em uma visualização em que os dados são separados das instruções numa mesma linha, cada linha agora representa toda a instrução com os dados. Isto pode ser visto na figura 4.3. Importante notar que é a mesma informação da figura 4.1, só que em outro formato visual.

Endereço	Instrução	Operando	Operando	Operando
0	1	54	2	
3	1	50	4	
6	7	46		
8	4	47	46	46
12	7	48		
14	4	49	50	48
18	3	51	47	49
22	6	52	51	
25	3	53	46	52
29	5	55	53	54
33	8	55		
35	2	56	46	52
39	5	57	56	54
43	8	57		
45	9			

Figura 4.3: Separando as instruções.

4.3.1 Abstração dos endereços

Continuando nossa exploração de aspectos percebidos pelo ser humano a partir da maneira como o computador trabalha, agora é possível percebermos mais duas coisas importantes:

1. O computador não mudou seu modo de operar, ele continua executando as instruções na memória conforme foi apresentado na seção 4.2.4;

2. A visualização na forma apresentada na figura 4.3 é apenas uma maneira mais simples para o ser humano perceber o que o computador está fazendo.

Esta segunda observação é muito importante. Ela nos permite aumentar o grau de "facilidade visual" ou, dizendo de outra maneira, o grau de *notação* sobre o modelo Von Neumann, o que vai nos permitir a compreensão do processo de uma maneira cada vez mais "humana".

De fato, a partir da figura 4.3, podemos perceber que o endereço em que ocorre a instrução é irrelevante para nós, humanos. Podemos perfeitamente compreender o processo de computação se eliminarmos a coluna do "Endereço" na figura 4.3.

A figura 4.4 ilustra como ficaria o programa em memória visto de outra maneira, agora não apenas em formato de blocos mas também sem a coluna dos endereços. Vamos perceber que, apesar de muito parecido com a figura anterior, o grau de "poluição visual" é bem menor.

Instrução	Operando	Operando	Operando
1	54	2	
1	50	4	
7	46		
4	47	46	46
7	48		
4	49	50	48
3	51	47	49
6	52	51	
3	53	46	52
5	55	53	54
8	55		
2	56	46	52
5	57	56	54
8	57		
9			

Figura 4.4: Abstração dos endereços.

Vale a pena reforçar: o computador não mudou, ele continua operando sobre os circuitos eletrônicos, o que estamos fazendo é uma tentativa, um pouco mais humana, de enxergar isto.

4.3.2 Abstração dos códigos das instruções

Continuando neste processo de "fazer a coisa do modo mais confortável para o ser humano", afirmamos que é possível aumentar ainda mais o grau de notação.

Para a etapa seguinte, vamos observar que, embora os computadores manipulem números (em particular, números binários) de maneira muito eficiente e rápida, o mesmo não ocorre para os humanos, que têm a tendência a preferirem *nomes*.

De fato, basta observar que nós usamos no nosso dia-a-dia nomes tais como Marcos, José ou Maria e não números tais como o RG do Marcos, o CPF do José ou o PIS da Maria.

Ora, já que é assim, qual o motivo de usarmos os números $1-9$ para as instruções se podemos perfeitamente usar o mnemônico associado ao código?

Desta forma, vamos modificar ainda mais uma vez nossa visualização da memória, desta vez escrevendo os nomes dos mnemônicos no lugar dos números. Assim, na

coluna *Instrução*, substituiremos o número 1 por `load`, 2 por `add`, 3 por `sub` e assim por diante, conforme a figura 4.2.

O programa da figura 4.4 pode ser visualizado novamente de outra forma, tal como apresentado na figura 4.5. Notem que o grau de compreensão do código, embora ainda não esteja em uma forma totalmente amigável, já é bastante melhor do que aquela primeira apresentação da "fotografia da memória".

De fato, agora é possível compreender o significado das linhas, em que existe um destaque para a operação (o mnemônico), a segunda coluna é o endereço de destino da operação e as outras colunas são os operandos.

Instrução	Operando	Operando	Operando
`load`	54	2	
`load`	50	4	
`read`	46		
`mult`	47	46	46
`read`	48		
`mult`	49	50	48
`sub`	51	47	49
`sqrt`	52	51	
`sub`	53	46	52
`div`	55	53	54
`write`	55		
`add`	56	46	52
`div`	57	56	54
`write`	57		
`stop`			

Figura 4.5: Programa reescrito com Mnemônicos.

O que falta ainda a ser melhorado? Nós humanos usamos desde a mais tenra idade outro tipo de notação para representarmos operações. Este é o próximo passo.

4.3.3 Abstração do repertório de instruções

Nesta etapa, observaremos que as instruções executadas pelo computador nada mais são do que manipulação dos dados em memória. Os cálculos são feitos sobre os dados e o resultado é colocado em alguma posição de memória.

Podemos melhorar o grau de abstração considerando a notação apresentada na figura 4.6. Isto é, vamos usar as tradicionais letras finais do alfabeto para ajudar na melhoria da facilidade visual. Assim poderemos reescrever o programa mais uma vez e ele ficará como apresentado na figura 4.7.

$x = [p+1]$
$y = [p+2]$
$z = [p+3]$

$[p]$	**Instrução**	**Notação**
1	`load` $x\ y$	$x \leftarrow [y]$
2	`add` $x\ y\ z$	$x \leftarrow [y] + [z]$
3	`sub` $x\ y\ z$	$x \leftarrow [y] - [z]$
4	`mult` $x\ y\ z$	$x \leftarrow [y] \times [z]$
5	`div` $x\ y\ z$	$x \leftarrow \frac{[y]}{[z]}$
6	`sqrt` $x\ y$	$x \leftarrow \sqrt{[y]}$
7	`read` x	$x \leftarrow V$
8	`write` x	$V \leftarrow [x]$
9	`stop`	$\bullet$

Figura 4.6: Notação para as instruções.

$$
\begin{array}{lcl}
54 & \leftarrow & 2 \\
50 & \leftarrow & 4 \\
46 & \leftarrow & \underline{\vee} \\
47 & \leftarrow & [46] \times [46] \\
48 & \leftarrow & \underline{\vee} \\
49 & \leftarrow & [50] \times [48] \\
51 & \leftarrow & [47] - [49] \\
52 & \leftarrow & \sqrt{[[51]]} \\
53 & \leftarrow & [46] - [52] \\
55 & \leftarrow & \frac{[53]}{[54]} \\
\square & \leftarrow & [55] \\
56 & \leftarrow & [46] + [52] \\
57 & \leftarrow & \frac{[56]}{[54]} \\
\square & \leftarrow & [57] \\
\bullet & &
\end{array}
$$

Figura 4.7: Programa escrito sob nova notação.

4.3.4 Abstração dos endereços de memória (variáveis)

Na verdade, assim como já observamos que o código da instrução não nos interessa, vamos perceber que o mesmo é verdade com relação aos endereços de memória. Então, convencionaremos que os números podem ser trocados por nomes. Fazemos isto na versão seguinte do mesmo programa.

Convém notar que o ser humano gosta de dar nomes apropriados para as coisas[4]. Assim, é importante que os nomes que usarmos tenham alguma relação com o

[4]Quem quiser ter uma ideia do estrago quando não damos bons nomes para as coisas, pode acompanhar dois diálogos humorísticos, um deles, o original, da dupla Abbot & Costello sobre um jogo de beisebol, a outra de um ex-presidente dos Estados Unidos falando sobre o novo presidente da China. Ambos os vídeos estão disponíveis no *YouTube*: (1) http://www.youtube.com/watch?v=sShMA85pv8M; (2) http://www.youtube.com/watch?v=Lr1DWkgUBTw.

significado que eles estão desempenhando no programa.

Então chega o momento de nós sabermos que o programa que está em memória recebe como entrada dois valores b e c e escreve como saída as raízes da equação $x^2 - bx + c = 0$. Os cálculos usam o método de Bhaskara. A figura 4.8 mostra a convenção para a substituição dos endereços por nomes.

Endereço	**Nome**
54	dois
50	quatro
46	B
47	quadradoB
48	C
49	quadruploC
51	discriminante
52	raizDiscriminante
53	dobroMenorRaiz
55	menorRaiz
56	dobroMaiorRaiz
57	maiorRaiz

Figura 4.8: Dando nomes para os endereços.

Agora, usaremos esta convenção de troca de endereços por nomes para podermos reescrever o programa ainda mais uma vez, obtendo a versão da figura 4.9.

Esta versão define o último grau de abstração simples. A partir deste ponto a notação deixa de ser só uma abreviatura das instruções e a tradução deixa de ser direta.

Apesar do fato de ser o último nível de notação simples, ainda é possível melhorarmos o grau de facilidade visual, mas desta vez passamos para a notação ou linguagem de "alto nível", que vai exigir a introdução dos chamados *compiladores*.

4.4 Abstração das instruções (linguagem)

Apesar de todas as notações e convenções que foram feitas no programa, até se chegar na versão mostrada na figura 4.9, de certa maneira, o programa ainda está em um formato muito parecido com o do programa original.

Para que seja possível aumentar o nível de notação ainda mais é preciso contar com a ajuda de *programas tradutores*, ou como eles são mais conhecidos, os *compiladores*.

Estes programas conseguem receber como entrada um texto escrito em um formato adequado e gerar como saída um programa no formato da máquina. Isto é

dois	$\leftarrow$	2
quatro	$\leftarrow$	4
B	$\leftarrow$	$\underline{\vee}$
quadradoB	$\leftarrow$	B $\times$ B
C	$\leftarrow$	$\underline{\vee}$
quadruploC	$\leftarrow$	quatro $\times$ C
discriminante	$\leftarrow$	quadradoB - quadruploC
raizDiscriminante	$\leftarrow$	$\sqrt{discriminante}$
dobroMenorRaiz	$\leftarrow$	B - raizDiscriminante
menorRaiz	$\leftarrow$	$\frac{dobroMenorRaiz}{dois}$
$\square$	$\leftarrow$	menorRaiz
dobroMaiorRaiz	$\leftarrow$	B + raizDiscriminante
maiorRaiz	$\leftarrow$	$\frac{dobroMaiorRaiz}{dois}$
$\square$	$\leftarrow$	maiorRaiz
$\bullet$		

Figura 4.9: Programa reescrito com nomes para variáveis.

possível apenas se os programas forem escritos em um formato que respeite regras extremamente rígidas, pois senão a tarefa não seria possível.

As linguagens de alto nível são definidas a partir de uma *gramática* extremamente mais rígida que a do português, por exemplo. Estas gramáticas, conhecidas como *gramáticas livre de contexto*, têm como uma de suas principais características que elas não permitem escrever programas ambíguos. O português permite.

Sem entrarmos muito em detalhes desta gramática, a título de exemplo mostraremos versões em mais alto nível do programa da figura 4.9. Estas versões são apresentadas na figura 4.10.

read B
read C
discriminante $\leftarrow$ B^2 - 4 $\times$ C
raizDiscriminante $\leftarrow \sqrt{discriminante}$
menorRaiz $\leftarrow \frac{B - raizDiscriminante}{2}$
write menorRaiz
maiorRaiz $\leftarrow \frac{B + raizDiscriminante}{2}$
write maiorRaiz

read B
read C
raizDiscriminante $\leftarrow \sqrt{B^2 - 4 \times C}$
write $\frac{\text{B - raizDiscriminante}}{2}$
write $\frac{\text{C + raizDiscriminante}}{2}$

Figura 4.10: Duas outras versões do programa.

Estas versões são compreensíveis para o ser humano, mas ainda não estão no formato ideal para servirem de entrada para o compilador, em particular por causa dos símbolos de fração ou do expoente. Os compiladores exigem um grau maior de rigidez, infelizmente. A disciplina Construção de Compiladores, no sexto período do curso, é exclusiva para o estudo profundo dos motivos desta dificuldade, tanto de se verificar se o programa está correto do ponto de vista gramatical, quanto do ponto de vista de se traduzir o programa para linguagem de máquina.

No momento, vamos nos limitar a apresentar na figura 4.11 uma versão do mesmo programa escrito em *Pascal*. Após compilação, o resultado é um programa que pode ser executado pelo computador.

Em suma, o compilador *Pascal* é um programa que, entre outras coisas, consegue transformar o código de alto nível mostrado na figura 4.11 e gerar um código que o computador possa executar tal como mostrado na primeira figura.

```
program bhaskara (input,output);
var b, c, raizdiscriminante: real;

begin
      read (b);
      read (c);
      raizdiscriminante:= sqrt(b*b - 4*c);
      write ((b - raizdiscriminante)/2);
      write ((b + raizdiscriminante)/2);
end.
```

Figura 4.11: Versão do programa escrito em *Pascal*.

4.5 Conclusão

Nesta parte do texto procuramos mostrar que qualquer linguagem de programação de alto nível (tal como *Pascal*, C ou JAVA) é meramente uma notação convencionada visando facilitar a vida do ser humano que programa o computador. Esta notação trata de como um *texto* se traduz em um *programa executável* em um determinado sistema operacional (que usa um determinado conjunto reduzido de instruções).

Um programa que traduz um texto que emprega uma certa notação convencionada em um programa executável é chamado de "compilador".

Assim, a arte de se programar um computador em alto nível é, basicamente, conhecer e dominar uma notação através da qual textos (ou *programas fonte*) são traduzidos em programas executáveis.

Programar, independentemente da linguagem utilizada, significa concatenar as instruções disponíveis dentro de um repertório a fim de transformar dados de entrada em dados de saída para resolver um problema.

Nas linguagens de alto nível, as instruções complexas são traduzidas em uma sequência de operações elementares do repertório básico da máquina. Por isto os programas fonte, quando compilados, geram executáveis que são dependentes do sistema operacional e do hardware da máquina onde o programa executa.

A partir destas ideias, partindo do princípio que se tem um algoritmo que resolve um problema, o que é preciso saber para se programar um computador?

- Ter à disposição um editor de textos, para codificar o algoritmo na forma de programa fonte;

- Ter à disposição um compilador para a linguagem escolhida (no nosso caso, o Free *Pascal*), para transformar automaticamente um programa fonte em um programa executável.

No restante deste curso, vamos nos preocupar com a arte de se construir algoritmos, tendo em mente que o estudante deverá ser capaz de saber transformar este algoritmo em forma de programa fonte de maneira que este possa ser compilado e finalmente executado em um computador.

4.6 Exercícios

1. Para perceber como o ambiente do computador é limitado em função do reduzido número de instruções disponíveis em baixo nível, você pode tentar jogar este jogo (`http://armorgames.com/play/6061/light-bot-20`). Nele, existe um boneco que tem que cumprir um percurso com o objetivo de apagar todas as células azuis do terreno quadriculado usando apenas poucos comandos e com pouca "memória" disponível. Você pode fazer o uso de duas funções que auxiliam nas tarefas repetitivas. Divirta-se!

2. Modifique a "fotografia da memória" apresentada para que o computador resolva a equação $ax^2 + bx + c = 0$ pelo método de Bhaskara. A diferença do que foi apresentado é o coeficiente a do termo x^2 e o sinal de b.

3. Leia os seguintes textos da *wikipedia*:

   (a) `http://pt.wikipedia.org/wiki/Arquitetura_de_von_Neumann`, sobre a arquitetura de von Neumann;

   (b) `http://pt.wikipedia.org/wiki/Von_Neumann`, sobre a vida de von Neumann, em especial a parte sobre computação.

Capítulo 5

Conceitos elementares

Agora que sabemos os princípios de algoritmos e as limitações da máquina, é preciso introduzir conceitos elementares de algoritmos, sem os quais não é possível seguir adiante.

Apresentaremos problemas simples o bastante para nos concentrarmos nas novidades, isto é, nos aspectos relevantes das estruturas de controle de fluxo e demais conceitos presentes nas linguagens de programação. Nos próximos capítulos estas estruturas elementares serão utilizadas na construção de soluções para problemas cada vez mais sofisticados.

5.1 Algoritmos e linguagens de programação

Conforme vimos, os algoritmos devem ser escritos em um nível de detalhamento suficiente para que o compilador consiga fazer a tradução do código para linguagem de máquina.

O compilador precisa receber um texto formatado em uma linguagem simples, não ambígua, precisa. Para isto as linguagens de programação seguem uma gramática rígida, se comparada com a da língua portuguesa. Também segue um vocabulário limitado, constituído de alguns poucos elementos.

O que vamos apresentar aqui é uma linguagem de mais alto nível do que aquela apresentada no capítulo 4. Trata-se de uma maneira de escrever que é um ponto intermediário entre a capacidade de redação do ser humano e a capacidade de compreensão do computador.

A base das linguagens de programação, em geral, é constituída por:

- a noção de *fluxo de execução de um programa*;
- os comandos da linguagem que modificam os fluxo de execução;
- os comandos, e demais conceitos da linguagem, que manipulam os dados em memória e a interação com o usuário (entrada e saída de dados);
- as expressões da linguagem que permitem a realização de cálculos aritméticos e lógicos;

Neste capítulo usaremos as regras do compilador *Free* Pascal e para isto o leitor deve ter em mãos algum guia de referência desta linguagem, por exemplo, o mini guia de referência que está disponível no site oficial da disciplina CI055[1], onde as explicações são detalhadas. Em sala de aula haverá explicação satisfatória. Por outro lado, os comandos da linguagem são suficientemente claros para que o programa faça sentido, basta traduzir literalmente os termos em inglês para suas versões em português.

Os códigos serão escritos em *Pascal*, pois acreditamos que editar código, compilar, executar e testar programas ajuda o aluno a compreender os conceitos teóricos. Desta forma os alunos poderão testar variantes em casa e melhorar o aprendizado.

5.2 Cálculo de raízes de uma equação do segundo grau

Problema: Calcular as raízes da equação do segundo grau $x^2 - bx + c = 0$.

No capítulo 4 nós seguimos em detalhes o processo de obtenção da solução em um modelo de baixo nível e chegamos a um código de alto nível escrito em *Pascal*. A figura 5.1 é uma cópia da solução apresentada na figura 4.11.

```
program bhaskara (input,output);
var b, c, raizdiscriminante: real;

begin
      read (b);
      read (c);
      raizdiscriminante:= sqrt(b*b - 4*c);
      write ((b - raizdiscriminante)/2);
      write ((b + raizdiscriminante)/2);
end.
```

Figura 5.1: Programa que implementa o método de Bhaskara.

Este código simples é rico em elementos das linguagens de programação. Ele contém quatro elementos importantes: os comandos de entrada e saída o comando de atribuição e as expressões aritméticas. Antes disso, relembramos que o fluxo de execução do programa é assim: o programa inicia após o *begin* e executa os comandos de cima para baixo, terminando no *end*.

Os dois primeiros comandos, *read*, servem para o usuário do programa fazer a carga dos valores dos coeficientes da equação para a memória do computador. Duas variáveis (abstrações para posições físicas de memória), a e b, recebem estes valores.

[1] http://www.inf.ufpr.br/cursos/ci055/pascal.pdf.

A linguagem *Pascal* exige a declaração dos tipos, no cabeçalho do programa (o que antecede o bloco entre o *begin* e o *end*. Isto é detalhado no material complementar sobre o compilador.

As duas últimas linhas contêm o comando *write*, que serve para imprimir alguma coisa na tela do usuário, neste caso, o resultado de um cálculo.

O cálculo propriamente dito é apresentado ao computador na forma de uma expressão aritmética, isto é, uma sequência de contas que são realizadas pelo computador: subtraia de b o valor calculado e armazenado na variável *raizdiscriminante* e em seguida divida tudo por 2. A regra para construção de expressões aritméticas é detalhado no material complementar.

A terceira linha de código ilustra um exemplo de um comando de atribuição, denotado pelo símbolo `:=`. O computador deve realizar o cálculo da expressão aritmética do lado direito do símbolo `:=` e somente após armazenar o valor resultante na variável que aparece do lado esquerdo.

Os cálculos são realizados conforme as regras descritas no material complementar: primeiro $b * b$ obtém o quadrado de b. Em seguida o valor de c é multiplicado por 4. O cálculo continua pela subtração do primeiro valor obtido pelo segundo. Finalmente a raiz quadrada deste último valor é calculada e apenas após tudo isto ocorrer o resultado é armazenado na variável *raizdiscriminante*.

Desta forma, expressões aritméticas servem para fazer cálculos. Comandos de entrada servem para o usuário fornecer dados ao computador. Comandos de saída servem para o usuário receber informações do computador. Comandos de atribuição servem para o computador manipular dados em memória. Estes são os elementos mais simples de qualquer programa.

5.3 Imprimir a soma de dois números dados

Vamos aqui considerar ainda um outro problema bem simples.

Problema: Ler dois números do teclado e imprimir a soma deles na tela.

O programa apresentado na figura 5.2 está correto e captura a excência da solução! Os comandos de leitura carregam os números digitados no teclado na memória, em posições acessíveis a partir dos nomes a e b. Em seguida, uma *expressão aritmética* faz o cálculo da soma de ambos e o resultado é impresso na tela.

Um pequeno problema é que, quando executado, o cursor fica piscando na tela e não deixa nenhuma mensagem sobre o que o usuário deve digitar.

O estudante pode querer modificar ligeiramente este código para produzir uma interface um pouco mais amigável, mas deve neste caso observar que o resultado será o mesmo. A versão minimamente modificada para este problema é apresentada na figura 5.3.

O programador iniciante deve ter em mente que não deve perder muito tempo com "firulas" na tela, pelo menos não neste curso. Em outras disciplinas, quando a arte da programação estiver dominada, o estudante aprenderá a integrar elegantemente uma

```
program soma2;
var a,b: integer;

begin
        read (a);
        read (b);
        write (a+b);
end.
```

Figura 5.2: Primeira solução.

```
program soma2;
var a,b: integer;

begin
        write ('entre com o valor de a: ');
        read (a);
        write ('entre com o valor de b: ');
        read (b);
        writeln (a,'+',b,'= ',a+b);
end.
```

Figura 5.3: Mesma solução, agora com interface amigável.

interface amigável com o usuário do programa ao mesmo tempo mantendo o código legível. Neste exemplo usamos a outra versão do comando de impressão, o *writeln*, que além de imprimir na tela muda o cursor de linha.

5.4 Imprimindo sequências de números na tela

Nesta seção usaremos como apoio um problema muito simples. Apesar disto, a discussão será bastante rica em conceitos de algoritmos.

Problema: Imprimir todos os números entre 1 e 5.

Provavelmente um humano escreveria algo assim:

1. imprima o número 1;
2. imprima o número 2;
3. imprima o número 3;
4. imprima o número 4;
5. imprima o número 5;

Este algoritmo pode ser codificado em *Pascal* tal como é ilustrado na figura 5.4, usando-se o comando de saída apresentado anteriormente.

```
program contar;

begin
        write (1);
        write (2);
        write (3);
        write (4);
        write (5);
end.
```

Figura 5.4: Primeira solução para contar de 1 a 5.

Após compilado e executado, os números 1, 2, 3, 4 e 5 aparecem na tela, atendendo ao enunciado. Mas, à luz das discussões do capítulo 2, é fácil ver que ela não é muito boa, pois não permite uma reimplementação simples de problemas similares. Por exemplo, se o enunciado fosse "Imprimir todos os números entre 1 e 20" no lugar de "todos entre 1 e 5", teríamos que escrever um código como o ilustrado na figura 5.5.

```
program contar;

begin
      write (1);
      write (2);
      write (3);
      write (4);
      write (5);
      write (6);
      write (7);
      write (8);
      write (9);
      write (10);
      write (11);
      write (12);
      write (13);
      write (14);
      write (15);
      write (16);
      write (17);
      write (18);
      write (19);
      write (20);
end.
```

Figura 5.5: Primeira solução modificada para números de 1 a 20.

Extrapolando o enunciado do problema, se fosse para imprimir números entre 1 e 30.000 ou entre 1 e 100.000.000, o programa ficaria com 30 mil ou 100 milhões de linhas de código extremamente repetitivo e de difícil e custosa edição.

A simplicidade do raciocínio inicial resultou em uma tarefa tão exaustiva que o computador deixou de ajudar, ele passou a atrapalhar!

Felizmente, há algoritmos melhores!

O computador é, conforme vimos, uma máquina com pouquíssimos recursos, mas que, se bem explorados, permite-nos escrever programas fantásticos. O raciocínio deve ser então desenvolvido de tal maneira que o trabalho exaustivo fique com o computador e não com o programador.

Primeira lição: não é perda de tempo pensar mais antes de escrever código!

As operações elementares da máquina são, basicamente, colocar e consultar dados da memória ou fazer contas com bastante rapidez. As operações são executadas uma por uma, em ordem, visualmente de cima para baixo.

Como explorar isto? Se pelo menos conseguíssemos uma versão em que o copiar/colar fosse viável, já ajudaria. A figura 5.6 ilustra uma solução em que isto é possível.

```
program contar;
var i: integer;

begin
        i:= 1;

        write (i);
        i:= i + 1;

        write (i);
        i:= i + 1;

        write (i);
        i:= i + 1;

        write (i);
        i:= i + 1;

        write (i);
        i:= i + 1;
end.
```

Figura 5.6: Segunda solução.

A expressão `i:= 1` é mais um exemplo de um *comando de atribuição*. O resultado da execução do comando é que a expressão à direita do símbolo `:=`, no caso o 1, é colocado na posição de memória acessada pela variável i.

A expressão `i + 1` é mais um exemplo de uma *expressão aritmética*, que é constituída conforme as regras do compilador. Uma expressão aritmética resulta em um valor numérico, no caso deste exemplo do mesmo tipo da variável i, que é *integer*.

Destacamos então que a linha de código contendo `i:= i + 1` também é uma atribuição e funciona assim: resolve-se o valor da expressão à direita do símbolo `:=`, isto é, o valor da variável i, que na primeira vez é 1, é somado com a constante 1, resultando no número 2. Em seguida, este valor por sua vez é colocado na posição da variável que aparece à esquerda do símbolo `:=`, casualmente a própria variável i, que passa a ter o valor 2.

O que ocorre é que uma certa variável i é iniciada com o valor 1 e sucessivamente então usa-se o comando de impressão para exibir o valor de i na tela, incrementando-se de 1 em 1 o valor da variável, obtendo-se como resultado final a impressão do valor 5 e sendo 6 o último valor (não impresso) da variável i.

O programa foi feito de maneira a ter blocos de comandos repetidos, pois ainda não sabemos mudar o fluxo de execução de um programa, isto é, ele executa "de cima para baixo", então a solução foi repetir os mesmos blocos 5 vezes.

É possível forçar a mudança do fluxo de execução do programa? O que precisamos é de uma estrutura que permita repetir um determinado trecho de código *enquanto uma determinada condição for verdadeira*.

Isto é conseguido com o uso de comandos de repetição. A linguagem *Pascal* oferece três maneiras de se fazer isto. Veremos apenas uma delas por enquanto. No decorrer do curso voltaremos às outras formas. A figura 5.7 ilustra o uso deste conceito.

```
program contar;
var i: integer;

begin
        i:= 1;
        while i <= 30000 do
        begin
              write (i);
              i:= i + 1;
        end;
end.
```

Figura 5.7: Sexta solução.

O comando de repetição executa os comandos aninhados no bloco entre o *begin* e o *end;* enquanto uma *expressão booleana* for verdadeira. No primeiro momento em que for falsa, o fluxo é alterado para depois do *end*.

A expressão $i \leq 30000$ é uma expressão booleana, retorna verdadeiro ou falso apenas, dependendo da avaliação dos valores pelo computador. No caso, vai resultar falso apenas quando i for estritamente maior do que 30000.

Neste exemplo, a variável i foi inicializada com o valor 1. Em seguida, os comandos de impressão e incremento são executados *enquanto* o valor de i for menor ou igual a 30000. Desta forma, o número 1 é impresso, i passa a valer 2, que é menor ou igual a 30000, então 2 é impresso e i passa a valer 3, que por sua vez ainda é menor ou igual a 30000. Então 3 é impresso na tela e i passa a valer 4, e assim por diante, até o momento em que será impresso na tela o valor 30000. Neste ponto i passará a valer 30001, que não é menor ou igual a 30000, por isto o fluxo de execução do programa vai para o comando que segue o bloco do comando *while*, isto é, o fim do programa.

Nesta seção mostramos o conceito de comando de repetição (*while/do*) e um exemplo de uso de uma expressões booleanas. A próxima seção apresentará a última estrutura básica que nos interessa.

5.5 Imprimir os quadrados de uma faixa de números

Ainda no contexto de problemas simples, este outro problema nos permite combinar as técnicas usadas nos problemas anteriores e resolver algo ligeiramente mais complicado.

Problema: Imprimir uma tabela com os valores de x e x^2 para todos os valores de x tais que $1 \leq x \leq 30$.

O programa que ilustra a solução para este problema é apresentado na figura 5.8 e imprime todos os valores *inteiros* de 1 a 30. Observamos que o enunciado não deixa claro, mas não seria possível imprimir todos os reais. Seria?

```
program quadrados;
var i: integer;

begin
        i:= 1;
        while i <= 30 do
        begin
              write (i,' ',i*i);
              i:= i + 1;
        end;
end.
```

Figura 5.8: Tabela com os quadrados dos números de 1 a 30.

O programa inicia com o valor de i igual a 1 e para cada valor entre 1 e 30 imprime na tela o próprio valor, seguido de um espaço em branco e finalmente o quadrado do número, calculado pela expressão aritmética `i*i`, que significa $i \times i$.

5.6 Imprimindo a soma de vários pares de números

Problema: Ler vários pares de números e imprimir a soma de cada par.

Este é uma variante do problema da seção 5.3. Naquele caso um único par de número era lido do teclado e a soma deles impressa. Agora temos que fazer a mesma coisa para vários pares de números dados como entrada.

A solução para um único par já estando feita, basta que o programa repita o mesmo trecho de código várias vezes, usando o comando *while/do*.

Apenas uma questão deve ser resolvida: quantos pares de números serão dados como entrada? O enunciado diz "vários pares", mas não estabelece o número preciso. Algoritmos não sabem lidar com isto. Fazendo uma analogia com o problema do bolo de chocolate, é como se o enunciado fosse "fazer um bolo". O cozinheiro não sabe que tipo de bolo está sendo solicitado e não consegue realizar a tarefa.

É necessário estabelecer uma condição de parada e isto deve estar claro no enunciado. Existem duas formas de se resolver isto:

- ou o enunciado estabelece a quantidade exata de números a serem lidos;
- ou o enunciado estabelece uma condição de parada alternativa.

No primeiro caso o enunciado deveria ser algo assim: "ler 30 pares de números do teclado e imprimir, para cada par, a soma deles".

No segundo caso poderia ser algo mais ou menos assim: "ler pares de números do teclado e imprimir, para cada par, a soma deles. O algoritmo deve parar a execução quando os dois números lidos forem iguais a zero".

A figura 5.9 ilustra as duas formas. A esquerda apresenta a solução usando-se um contador, a da direita apresenta a solução com critério de parada alternativo.

```
program soma2variasvezes_v1;
var a,b,cont: integer;
(* cont conta os numeros lidos *)
begin
      cont:= 1;
      while cont <= 30 do
      begin
            read (a);
            read (b);
            writeln (a+b);
            cont:= cont + 1;
      end;
end.
```

```
program soma2variasvezes_v2;
var a,b: integer;

begin
      read (a);
      read (b);
      while (a <> 0) or (b <> 0) do
      begin
            writeln (a+b);
            read (a);
            read (b);
      end;
end.
```

Figura 5.9: Duas formas para somar pares de números.

Este exemplo é extremamente interessante pois permite ver claramente que o código referente aos comandos de leitura e impressão são os mesmos, apesar da ordem diferente no segundo exemplo. O que difere é a estrutura de controle do *laço*, isto é, do bloco de comandos que se repete. No quadro da esquerda, o algoritmo precisa contar até 30, por isto existe uma variável *cont* que faz esta conta. No quadro da direita, o laço é controlado por uma expressão booleana um pouco mais sofisticada que a do exemplo anterior, pois faz uso do operador *or*. A expressão em questão se torna falsa apenas quando ambos os números lidos forem nulos. Se um dois dois for não nulo, o laço é executado.

O comando de impressão ficou aparentemente invertido pela simples razão de que o teste depende de uma primeira leitura das variáveis *a* e *b antes do teste*, senão o teste não pode ser feito. Mas, ressaltamos, o núcleo do programa é exatamente o mesmo, lê dois números e imprime a soma, o que muda é o controle do laço.

5.7 Testar se um número lido é positivo

Problema: Ler um único número do teclado e imprimí-lo apenas se ele for positivo.

Parece simples para o ser humano saber se um número é positivo ou não, mas

para o computador o que está em memória é apenas uma sequência de bits. Logo, para ele decidir se um número é positivo deve usar uma expressão booleana. Mas, como executar o comando de impressão apenas em um caso (do número ser positivo) e ignorar a impressão caso não seja? Este problema permitirá introduzir a noção de *desvio condicional*.

Um desvio condicional produz exatamente o efeito desejado, faz um teste e dependendo do resultado executa ou não o comando subsequente. A figura 5.10 ilustra a solução deste problema. No caso, o comando *writeln* só é executado se a expressão booleana for verdadeira. Caso o número lido seja nulo ou negativo o fluxo de execução do programa "pula" para o comando subsequente, que no caso é o fim do programa.

```
program imprime_se_positivo;
var a,b: integer;

begin
      read (a);
      if a > 0 then
            writeln (a); (* so executa se a for positivo *)
end.
```

Figura 5.10: Imprime se for positivo.

O comando de desvio condicional admite uma segunda forma, que estabelece um fluxo alternativo ao programa dependendo do teste.

Considere o seguinte problema.

Problema: Ler um único número do teclado e imprimí-lo apenas se ele for positivo. Se não for imprimir a mensagem "número inválido".

A solução deste problema requer uma variante do comando *if*, conforme ilustrado na figura 5.11. Apenas um comando de impressão será executado.

```
program imprime_se_positivo_v2;
var a,b: integer;

begin
      read (a);
      if a > 0 then
            writeln (a)                    (* so executa se a for positivo *)
      else
            writeln ('numero invalido'); (* executa se a for nulo ou negativo *)
end.
```

Figura 5.11: Imprime se for positivo, segunda versão.

5.8 Resumo

Neste capítulo vimos todas as estruturas elementares para qualquer algoritmo (ou programa) presentes em qualquer linguagem de programação. São eles:

Comandos

- Entrada e saída (*read* e *write*, respectivamente);
- Atribuição (:=);
- Repetição (*while/do*);
- Desvio condicional (*if/then*, ou *if/then/else*);

Expressões

- Aritméticas;
- Booleanas.

Qualquer algoritmo é escrito como uma combinação destes comandos manipulando dados em memória (variáveis) da maneira como o algoritmo estabelece. Como sabemos, cada problema tem várias soluções possíveis, mas uma vez fixado uma, pode-se escrever o programa na forma como o computador entenda, usando-se apenas as noções apresentadas neste capítulo.

A menos do uso de estruturas de dados sofisticadas, as quais veremos a partir do capítulo 10, agora já é possível escrever qualquer algoritmo, e consequentemente, qualquer programa!

O que muda de uma linguagem de programação para outra é basicamente a grafia dos comandos, as vezes com alguma ligeira modificação na sintaxe e na semântica do comando.

Por exemplo, na linguagem *C*, o comando *write* é grafado *printf*, na linguagem *BASIC* é grafado *print*. Cada linguagem tem um comportamento diferente, não é apenas o nome que muda. Por exemplo, se imprime e muda de linha ou não, em qual formato escreve, etc.

Mas, estes são os elementos básicos para se programar. No próximo capítulo veremos como usar estes comandos de maneira inteligente para resolver problemas mais complexos. O leitor pode pular o apêndice da próxima seção sem prejuízo da compreensão do restante do texto.

5.9 Apêndice

Existe mais um comando de controle de fluxo, presente em qualquer linguagem de programação: o *comando de desvio incondicional*. Segundo o modelo de Von Neumann estudado, isto nada mais é do que alterar o controlador de instruções para um endereço arbitrário (na verdade controlado) quando pensamos no modelo de baixo nível.

A figura 5.12 ilustra o uso do desvio incondicional para resolver o problema de se imprimir todos os números de 1 a 30.

```
program contar;
label: 10;
var i: integer;

begin
        i:= 1;

    10:write (i);
        i:= i + 1;

        if i <= 30 then
            goto 10;
end.
```

Figura 5.12: Exemplo do uso do desvio incondicional.

Neste exemplo, a variável i é inicializada com o valor 1 e em seguida é executado o comando que imprime 1 na tela. Em seguida a variável i é incrementada e, após verificar que 1 é menor ou igual a 30, o fluxo do programa executa o comando de desvio incondicional *goto*, que faz com que o fluxo de execução do programa vá para a linha indicada pelo rótulo 10, isto é, imprime o valor da variável i, incrementa o i e assim por diante. Quando i valer 31 o *goto* não é executado, o que faz com que o *end* final seja atingido e o programa termina.

Em outras palavras, é uma outra maneira (mais estranha) de se implementar o mesmo programa da figura 5.7. A observação interessante é que, no nível de máquina, o que o compilador faz é gerar a partir do programa da figura 5.7, um código de máquina implementado usando-se uma noção de desvio incondicional, implementada no repertório reduzido de instruções, mas isto o programador não é obrigado a saber agora.

O que ele deve saber é que o uso de comandos de desvio incondicional não é recomendado pois na medida em que os programas vão ficando com muitas linhas de código, digamos algumas centenas de linhas ou mais, o que ocorre é que o programador tende a se perder e passa a ter muita dificuldade em continuar o desenvolvimento do programa quando o uso do *goto* é feito de qualquer maneira e, em especial, de forma exagerada. Isto tende a tornar o código ilegível e de difícil manutenção.

Como vimos, as linguagens de alto nível permitem mecanismos mais elegantes para repetir trechos de código sem necessidade do uso do *goto*, usando-se o *while*. Isto é baseado no princípio da *programação estruturada*, que nasceu com a linguagem *ALGOL-60*. Esta linguagem, apesar de não ser mais usada, deu origem à maior parte das linguagens modernas, entre elas o próprio *Pascal*.

Este comando está em um apêndice pois nós nunca mais o usaremos.

5.10 Exercícios

1. Baixe o mini guia da linguagem *Pascal*, disponível em `http://www.inf.ufpr.br/cursos/ci055/pascal.pdf`. Você vai precisar estudá-lo para poder resolver alguns dos exercícios deste capítulo. Estude com atenção os seguintes capítulos: de 1 a 3; capítulo 5, até a sessão 5.5 (pule de 5.6 em diante); capítulos 6 e 7.

2. Para cada uma das expressões aritméticas abaixo, determine o tipo de dados da variável que está no lado esquerdo do comando de atribuição bem como o resultado da expressão que está no lado direito:

   (a) $A := 1 + 2 * 3;$
   (b) $B := 1 + 2 * 3/7;$
   (c) $C := 1 + 2 * 3\ DIV\ 7;$
   (d) $D := 3\ DIV\ 3 * 4.0;$
   (e) $E := A + B * C - D$

3. Indique qual o resultado das expressões abaixo, sendo:
   a=6; b=9.5; d=14; p=4; q=5; r=10; z=6.0 ; sim= TRUE

   (a) `((z/a)+b*a)-d DIV 2`
   (b) `p*(r mod q)-q/2`
   (c) `NOT sim AND (z DIV y + 1 = x)`
   (d) `(x + y > z) AND sim OR (y >= x)`

4. Indique qual o resultado das expressões abaixo, sendo:
   a=5; b=3; d=7; p=4; q=5; r=2; x=8; y=4; z=6; sim=TRUE

   (a) `(z DIV a + b * a) - d DIV 2`
   (b) `p / r mod q - q / 2`
   (c) `(z DIV y + 1 = x) AND sim OR (y >= x)`

5. Escreva em *Pascal* as seguintes expressões aritméticas usando o mínimo possível de parênteses:

   (a) $\frac{W^2}{Ax^2+Bx+C}$
   (b) $\frac{\frac{P_1+P_2}{Y-Z}R}{\frac{W}{AB}+R}$

6. Para compilar este programa, descubra os erros e os corrija.

```
(* programa que le um numero e retorna o ultimo algarismo *)
(* escrito por Marcos Castilho em 22/10/2002, com erros.  *)
program ultalgarismo;
begin
  read (A)
  writeln (A mod 10);
end.
```

7. Seja o seguinte programa em *Pascal*:

```
program Misterio;
var nota1,nota2,media: integer;
begin
      readln(nota1,nota2)
      while nota1 <> 0 do
            media:=nota1+nota2/2;
            writeln(nota1,nota2,media);
            readln(nota1,nota2);
end.
```

   (a) Quantos são, e quais são, os erros de compilação deste programa?

   (b) Considerando que você corrigiu os erros de compilação corretamente, o que faz este programa?

   (c) Considerando a estrutura do programa, os nomes das variáveis e a endentação usada, podemos afirmar que o programador cometeu alguns erros de lógica. Quantos são e quais são estes erros?

   (d) O que faz o programa faz após estas correções?

8. Dado o programa abaixo, mostre o acompanhamento de sua execução para três valores de entrada (valores pequenos). Em seguida, descreva o que o programa faz, comprovando suas afirmações.

```
program questao1(input, output);
var
  x: integer;
  y, m: longint;
begin
  read(x);
  y := 0;
  m := 1;
  while x > 0 do
    begin
      y := y + (x mod 2) * m;
      x := x div 2;
      m := m * 10;
    end;
  writeln(y)
end.
```

9. Dado o programa abaixo, mostre o acompanhamento de sua execução para três valores de entrada (valores pequenos). Em seguida, descreva o que o programa faz, comprovando suas afirmações.

```
program questao1(input, output);
var
  x: integer;
  y, m: longint;
begin
  read(x);
  y := 0;
  m := 1;
  while x > 0 do
    begin
      y := y + (x mod 2) * m;
      x := x div 2;
      m := m * 10;
    end;
  writeln(y)
end.
```

10. Considere o seguinte código fonte escrito em *Pascal*:

```
program prova (input,output);
var
  i, j, VAL, N: integer;
begin
  for i:= 1 to 4 do
  begin
        read (VAL);
        writeln (VAL, i);
        for j:= 3 to 5 do
```

```
        begin
              read (VAL);
              N:= VAL + i -j;
              writeln (VAL,j,N);
        end;
        read (VAL);
    end;
end.
```

Suponha que você dê como entrada de dados uma sequência crescente 1, 2, 3, 4, ..., na medida em que forem sendo executados os comandos "read". Qual a saída que será mostrada na tela do computador?

11. Tente compilar e executar com várias entradas os programas *Pascal* vistos neste capítulo. Uma sugestão é você copiar o programa fonte do PDF da apostila e colar em um editor de textos ASCII.

12. Faça um programa em *Pascal* que some duas horas. A entrada deve ser feita lendo-se dois inteiros por linha, em duas linhas, e a saída deve ser feita no formato especificado no exemplo abaixo:

```
Entrada:
12 52
7 13
```

```
Saída:
12:52 + 7:13 = 20:05
```

13. Dadas duas frações ordinárias a/b e c/d, determinar a sua soma e o seu produto, no formato de frações. A entrada de dados deve ser constituída de duas linhas, cada uma contendo dois inteiros, uma para a e b outra para c e d. A saída deverá ser também de duas linhas cada uma contendo um par que representa o numerador e o denominados da soma e produto calculadas. Exemplo para as frações $\frac{2}{5}$ e $\frac{7}{3}$:

```
Entrada:
2 5
7 3
```

```
Saída:
Soma: 41 15
Produto: 14 15
```

14. Dados o primeiro termo e a razão de uma progressão aritmética, determinar a soma dos seus primeiros cinco termos.

15. Dados dois números reais positivos determinar o quociente inteiro do primeiro pelo segundo usando apenas os operadores aritméticos reais.

16. Dado um número real positivo determinar sua parte inteira e sua parte fracionária usando apenas os operadores aritméticos reais.

17. Dado um número inteiro que representa uma quantidade de segundos, determinar o seu valor equivalente em graus, minutos e segundos. Se a quantidade de segundos for insuficiente para dar um valor em graus, o valor em graus deve ser 0 (zero). A mesma observação vale em relação aos minutos e segundos. Por exemplo: 3.600 segundos = 1 grau, 0 minutos, 0 segundos. ; 3.500 segundos = 0 graus, 58 minutos e 20 segundos.

```
Entrada:                Saída:
3600                    1, 0, 0
Entrada:                Saída:
3500                    0, 58, 20
```

18. Fazer um programa em *Pascal* para ler do teclado um número inteiro m e em seguida uma sequência de m números reais e imprimir a média aritmética deles. Isto é, dados os números $N_1, N_2, \ldots, N_m$, calcular:

$$\frac{N_1 + N_2 + \ldots + N_m}{m}$$

19. Fazer um programa em *Pascal* para calcular o produto dos números ímpares de A até B, onde $A \leq B$ são lidos do teclado. Considere que A e B são sempre ímpares. Isto é, calcular:

$$A \times (A+2) \times (A+4) \times \ldots \times B$$

20. Fazer um programa em *Pascal* para calcular o valor da soma dos quadrados dos primeiros 50 inteiros positivos não nulos.

$$\sum_{i=1}^{50} i^2 = 1^2 + 2^2 + 3^2 + \ldots + 50^2$$

21. Ler um inteiro positivo N diferente de zero e calcular a soma: $1^3 + 2^3 + ... + N^3$.

22. Fazer um programa em *Pascal* para ler uma massa de dados onde cada linha da entrada contém um número par. Para cada número lido, calcular o seu sucessor par, imprimindo-os dois a dois em listagem de saída. A última linha de dados contém o número zero, o qual não deve ser processado e serve apenas para indicar o final da leitura dos dados. Exemplo:

```
Entrada:
12                      Saída:
6                       14
26                      8
86                      28
0                       88
```

23. Fazer um programa em *Pascal* para ler uma massa de dados onde cada linha contém dois valores numéricos sendo o primeiro do tipo real e o segundo do tipo inteiro. O segundo valor é o peso atribuído ao primeiro valor. O programa deve calcular a média ponderada dos diversos valores lidos. A última linha de dados contém os únicos números zero. Esta linha não deve ser considerada no cálculo da média. Isto é, calcular o seguinte, supondo que m linhas foram digitados:

$$\frac{N_1 \times P_1 + N_2 \times P_2 + \ldots + N_m \times P_m}{P_1 + P_2 + \ldots P_m}$$

```
Entrada:
60 1
30 2
40 3
0 0
```

```
Saída:
40
```

24. Fazer um programa em *Pascal* que, dados dois números inteiros positivos, determine quantas vezes o primeiro divide exatamente o segundo. Se o primeiro não divide o segundo o número de vezes é zero. Por exemplo, 72 pode ser dividido exatamente por 3 duas vezes.

```
Entrada:
72 3
```

```
Saída:
2
```

25. Fazer um programa em *Pascal* para ler uma massa de dados contendo a definição de várias equações do segundo grau da forma $Ax^2 + Bx + C = 0$. Cada linha de dados contém a definição de uma equação por meio dos valores de A, B e C do conjunto dos números reais. A última linha informada ao sistema contém 3 (três) valores zero (exemplo 0.0 0.0 0.0). Após a leitura de cada linha o programa deve tentar calcular as duas raízes da equação. A listagem de saída, em cada linha, deverá conter sempre os valores de A, B e C lidos, seguidos dos valores das duas raízes reais. Considere que o usuário entrará somente com valores A, B e C tais que a equação garantidamente tem duas raízes reais. Note que você deve usar a fórmula de Bhaskara completa, incluindo o coeficiente A do termo x^2, já que na figura 4.11 o coeficiente de A era 1.

```
Entrada:
1.00 -1.00 -6.00
1.00 0.00 -1.00
0.00 0.00 0.00
```

```
Saída:
1.00 -1.00 -6.00 -3.00 2.00
1.00 0.00 -1.00 -1.00 1.00
```

26. Fazer um programa em *Pascal* que receba dois números inteiros N e M como entrada e retorne como saída $N \ mod \ M$ (o resto da divisão inteira de N por M) usando para isto apenas operações de subtração. O seu programa deve considerar que o usuário entra com N sempre maior do que M.

```
Entrada:
3 2
```

```
Saída:
1
```

27. Fazer um programa em *Pascal* que receba uma massa de dados contendo o saldo bancário de alguns clientes de um banco e imprima aqueles que são negativos. O último saldo, que não corresponde a nenhum cliente (e portanto não deve ser impresso), contém o valor zero.

```
Entrada:
832.47
215.25
-1987.11
19.00
-45.38
0
```

```
Saída:
-1987.11
-45.38
```

28. (*) Uma agência governamental deseja conhecer a distribuição da população do país por faixa salarial. Para isto, coletou dados do último censo realizado e criou um arquivo contendo, em cada linha, a idade de um cidadão particular e seu salário. As idades variam de zero a 110 e os salários variam de zero a 19.000,00 unidades da moeda local (salário do seu dirigente máximo). Considere o salário mínimo igual a 450,00 unidades da moeda local.

    As faixas salariais de interesse são as seguintes:

    - de 0 a 3 salários mínimos
    - de 4 a 9 salários mínimos
    - de 10 a 20 salários mínimos
    - acima de 20 salários mínimos.

    Fazer um programa em *Pascal* que leia o arquivo de entrada e produza como saída os percentuais da população para cada faixa salarial de interesse. A última linha, que não deve ser processada, contém dois zeros.

```
Entrada:
25 240.99
48 2720.77
37 4560.88
34 19843.33
23 834.15
90 315.87
78 5645.80
44 150.33
56 2560.00
67 2490.05
0 0.00
```

```
Saída:
4%
3%
2%
1%
```

29. (*) Escrever um programa em *Pascal* que leia do teclado uma sequência de números inteiros até que seja lido um número que seja o dobro ou a metade do anteriormente lido. O programa deve imprimir na saída os seguintes valores:

    - a quantidade de números lidos;
    - a soma dos números lidos;
    - os dois valores lidos que forçaram a parada do programa.

    Exemplo 1:

```
Entrada:
-549 -716 -603 -545 -424 -848
Saída:
6 -3685 -424 -848
```

Exemplo 2:

```
Entrada
-549 -716 -603 -545 -424 646 438 892 964 384 192
Saída
11 679 384 192
```

30. (*) Aqui temos uma forma peculiar de realizar uma multiplicação entre dois números: multiplique o primeiro por 2 e divida o segundo por 2 até que o primeiro seja reduzido a 1. Toda vez que o primeiro for impar, lembre-se do segundo. Não considere qualquer fração durante o processo. O produto dos dois números é igual a soma dos números que foram lembrados. Exemplo: $53 \times 26 =$

53	26	13	6	3	1
26	52	104	208	416	832
26 +		104 +		416 +	832 = 1378

Fazer um programa em *Pascal* que receba dois números inteiros e retorne o produto deles do modo como foi especificado acima.

```
Entrada:
53
26
```

```
Saída:
1378
```

31. (***) Desafio: Fazer um programa em *Pascal* que troque o conteúdo de duas variáveis inteiras sem utilizar variáveis auxiliares. Exemplo:

```
Entrada:
3 7
```

```
Saída:
7 3
```

Capítulo 6

Técnicas elementares

O objetivo deste capítulo é o domínio por parte do estudante das principais estruturas de controle de fluxo, em particular quando usadas de maneira combinada: atribuição; entrada e saída; desvio condicional e repetição, além do uso de expressões.

6.1 Atribuições dentro de repetições

Nesta seção veremos exemplos de problemas cujas soluções algorítmicas requerem o uso de atribuições aninhadas em repetições.

6.1.1 Somando números

O problema abaixo servirá para discutirmos várias maneiras de se resolver um mesmo problema até chegarmos na maneira mais elegante. Aproveitamos para discutir a técnica dos acumuladores.

Problema: Ler 5 números positivos do teclado e imprimir a soma deles.

A primeira e mais simples solução consiste em ler os 5 números do teclado e em seguida imprimir a soma deles. O programa da figura 6.1 mostra o algoritmo implementado para esta primeira proposta.

```
program soma_valores;
var
      a1, a2, a3, a4, a5:  integer;

begin
      read (a1, a2, a3, a4, a5);
      writeln (a1 + a2 + a3 + a4 + a5);
end.
```

Figura 6.1: Primeira solução.

O programa funciona e atende ao requisitado no enunciado. Mas a solução não é boa. Imagine que fosse pedido a soma não de 5 números, mas de 50. Para manter a mesma ideia do algoritmo anterior, teríamos que escrever algo como ilustrado na figura 6.2.

```
program soma_valores_50;
var
    a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17,
    a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28, a29, a30, a31, a32,
    a33, a34, a35, a36, a37, a38, a39, a40, a41, a42, a43, a44, a45, a46, a47,
    a48, a49, a50:  integer;

begin
    read (a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15,
          a16, a17, a18, a19, a20, a21, a22, a23, a24, a25, a26, a27, a28,
          a29, a30, a31, a32, a33, a34, a35, a36, a37, a38, a39, a40, a41,
          a42, a43, a44, a45, a46, a47, a48, a49, a50);

    writeln (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10 + a11 + a12 +
             a13 + a14 + a15 + a16 + a17 + a18 + a19 + a20 + a21 + a22 + a23 +
             a24 + a25 + a26 + a27 + a28 + a29 + a30 + a31 + a32 + a33 + a34 +
             a35 + a36 + a37 + a38 + a39 + a40 + a41 + a42 + a43 + a44 + a45 +
             a46 + a47 + a48 + a49 + a50);
end.
```

Figura 6.2: Imprimindo a soma de 50 números lidos no teclado.

É óbvio que esta técnica não é adequada. Além disto, só pode ser usada quando se sabe quantos números serão fornecidos. Para o problema abaixo, o algoritmo não pode ser usado.

Problema: Ler uma sequência de números positivos do teclado e imprimir a soma deles. O programa deve terminar quando o número lido do teclado for zero.

Como usar uma variável para cada valor lido? Uma vez que não se sabe, a princípio, quantos números serão digitados, não há como fazer isto. Logo, a técnica de solução não pode ser esta.

O algoritmo apresentado na figura 6.3 faz uso de uma técnica elementar em programação que é o uso de *acumuladores*.

Convém observar que a parte do algoritmo relativa a entrada de dados controlada pelo laço já foi estudada no capítulo anterior. Desta forma, se eliminarmos todas as linhas que contém a variável *soma* o que teremos é um programa que lê vários números do teclado até que seja lido um zero. O programa não faria nada com os números lidos.

Usa-se uma variável (soma) cujo papel é acumular valores, isto é, a variável é inicializada com um valor nulo e na sequência, ela é atualizada para os outros valores à medida em que eles são lidos do teclado.

A inicialização do acumulador com o valor zero deve-se ao fato deste ser o elemento

neutro da adição. Se fosse uma multiplicação de vários números, o acumulador deveria ser inicializado com o elemento neutro da multiplicação, isto é, o 1.

```
program soma_valores;
var
      numero, soma:   integer;

begin
      soma:= 0;                        (* inicializa o acumulador *)
      read (numero);
      while numero <> 0 do
      begin
            soma:= soma + numero;      (* atualiza o acumulador *)
            read (numero);
      end;
end.
```

Figura 6.3: Técnica do acumulador.

Na figura 6.4 mostra-se como resolver o problema inicial usando a técnica dos acumuladores. Notem que a solução explora um comando de atribuição sob o escopo de um comando de repetição.

Trocando-se o valor da constante MAX de 5 para 50, tem-se a solução para o problema 2. É fácil perceber que esta maneira é genérica, resolve problemas similares para qualquer tamanho de entrada, bastando-se trocar o valor da constante MAX.

```
program soma_valores;
const max=5;
var
      numero, i, soma:   integer;

begin
      soma:= 0;
      i:= 1;
      while i <= max do
      begin
            read (numero);
            soma:= soma + numero;
      end;
end.
```

Figura 6.4: Solução para o primeiro problema com acumuladores.

6.2 Desvios condicionais aninhados

O problema a seguir nos permitirá apresentar um algoritmo que o resolve de maneira elegante através do uso de estruturas contendo aninhamentos de desvios condicionais.

6.2.1 O menor de três

Problema: Após ler três números no teclado, imprimir o menor deles.

A solução para este problema (figura 6.5) envolve uma série de tomada de decisões com diversos fluxos alternativos de código.

Quando um comando está sob o controle de outro, diz-se que o comando mais interno está *sob o escopo* do mais externo. Neste caso, vamos usar um desvio condicional que controla outro. No jargão dos programadores, se diz que os comandos nesta forma estão *aninhados*.

```
program imprime_menor;
var
    a, b, c:  integer;

begin
     write('entre com tres numeros inteiros: ');
     read(a, b, c);
     if a < b then
          if a < c then
               writeln ('o menor dos tres eh ',a)
          else
               writeln ('o menor dos tres eh ',c)
     else
          if b < c then
               writeln ('o menor dos tres eh ',b)
          else
               writeln ('o menor dos tres eh ',c)
end.
```

Figura 6.5: Imprimir o menor dentre 3 números lidos.

6.3 Desvios condicionais dentro de repetições

Nesta seção veremos exemplos de problemas cujas soluções algorítmicas requerem o uso de atribuições aninhadas em repetições.

6.3.1 Imprimir apenas números positivos

Neste problema estudaremos como aninhar um comando de desvio condicional dentro do escopo de um comando de repetição. Também aproveitaremos para iniciar o estudo sobre como pensar no problema global e nos possíveis subproblemas existentes.

Problema: Ler do teclado 30 números inteiros e imprimir na tela aqueles que são

positivos, ignorando os negativos ou nulos.

Se o enunciado fosse simplesmente para ler e imprimir 30 números, como faríamos? Provavelmente como ilustrado na figura 6.6. Observamos que este problema é muito similar a outros já estudados no capítulo anterior. Trata-se de um laço que precisa de um contador para se determinar a saída.

```
program lereimprimir;
var i, a: integer; (* i serve para contar quantos numeros foram lidos *)

begin
        i:= 1;
        while i <= 30 do
        begin
              read (a);
              writeln (a);
              i:= i + 1;
        end;
end.
```

Figura 6.6: Lendo e imprimindo 30 números.

O problema é que não queremos imprimir todos os números lidos, mas apenas aqueles que são positivos: *se* o número lido for positivo, então queremos imprimir, senão não. Conforme já visto, o comando *if* faz exatamente isto, testa uma expressão booleana que, caso satisfeita, executa o comando subsequente, senão o pula, passando diretamente para o seguinte, tal como ilustrado na figura 6.7.

```
program lereimprimirpositivos;
var i, a: integer;

begin
        i:= 1;
        while i <= 30 do
        begin
              read (a);
              if a > 0 then
                    writeln (a); (* so eh executado quando a eh positivo *)
              i:= i + 1;
        end;
end.
```

Figura 6.7: Lendo e imprimindo os positivos apenas.

Relembrando, o comando de desvio condicional também permite executar exclusivamente uma ação alternativa, caso o teste da expressão booleana resulte em falso. Por exemplo, se o enunciado fosse "ler 30 números e imprimir os que são pares mas imprimir o quadrado dos que não são, incluindo o zero".

Do ponto de vista lógico, seria o mesmo que dizer o seguinte: "*se* o número lido for positivo, imprimí-lo, *caso contrário* ele é zero ou negativo, então imprimir o quadrado dele". Isto pode ser visto na figura 6.8. Destacamos mais uma vez que a estrutura de controle do laço não mudou, apenas os comandos que são executados no laço mudaram, caracterizando-se dois subproblemas: um para controlar o laço, outro para se decidir se o número é positivo ou não e o que fazer com ele.

```
program lereimprimirpositivosequadrados;
var i, a: integer;

begin
        i:= 1;
        while i <= 30 do
        begin
              read (a);
              if a > 0 then
                    writeln (a) (* so eh executado quando a for positivo *)
              else
                    writeln (a*a); (* so eh executado quando a<= 0 *)
              i:= i + 1;
        end;
end.
```

Figura 6.8: Lendo e imprimindo os positivos e os quadrados dos ímpares.

Agora, o comando que imprime a só é executado quando $a > 0$. Se isto não for verdade, o que é impresso é o quadrado de a. Em ambos os casos o valor de i é incrementado.

Este último programa pode facilmente ser modificado para se resolver o problema de se contar quantos números lidos são positivos e quantos não são. Basta, ao invés de imprimir, contar, ou em outras palavras, acumular.

Para isto são necessárias duas variáveis adicionais, que iniciam em zero e que são incrementadas a cada vez que um número é identificado como positivo ou não. Esta ideia pode ser vista na figura 6.9. Importante notar a similaridade das duas soluções. Os problemas são ligeiramente diferentes, eles diferem em um dos subproblemas apenas: um deve imprimir, o outro deve acumular. Os outros problemas tais como o controle do laço e a leitura dos dados são exatamente os mesmos problemas.

Podemos usar o mesmo raciocínio para resolver toda uma classe de problemas similares, isto é, imprimir a soma de números que satisfazem alguma propriedade.

No caso do problema anterior, a propriedade era "ser negativo" ou "ser positivo". Mas poderia ser qualquer outra coisa, como por exemplo "ser diferente de zero", "ser primo", "ser múltiplo de 50" ou algo mais complexo como "ter saldo positivo nos últimos 12 meses e não estar devendo para o imposto de renda".

Vejamos um problema similar para uma propriedade simples, cuja solução pode ser facilmente adaptada do algoritmo anterior.

```
program contarpositivosnegativosenulos;
var i,                (* serve para contar ate 30          *)
    conta_positivos,  (* serve para contar os positivos     *)
    conta_outros,     (* serve para contar os nao positivos *)
    a: integer;       (* numeros lidos na entrada           *)

begin
        conta_positivos:= 0; (* eh preciso inicializar a variavel *)
        conta_outros:= 0;
        i:= 1;
        while i <= 30 do
        begin
             read (a);
             if a > 0 then
                  conta_positivos:= conta_positivos + 1
             else
                  conta_outros:= conta_outros + 1;
             i:= i + 1;
        end;
        writeln ('A quantidade de positivos lidos eh ',conta_positivos);
        writeln ('A quantidade de nao positivos lidos eh ',conta_outros);
end.
```

Figura 6.9: Contando os positivos e os negativos e nulos.

6.3.2 Somando pares e ímpares

Problema: Ler uma sequência de números e imprimir separadamente a soma dos que são *pares* e a soma dos que são ímpares. O programa deve terminar quando o número lido for o zero. Este último número também deve ser ignorado.

No programa ilustrado na figura 6.10, basicamente foi colocado no lugar do *if* $x > 0$ *then* a tradução da expressão *if "x é par" then*, tradução esta que envolve o uso da operação de *resto de divisão inteira*. Todo número que, dividido por 2, resulta em resto 0 só pode ser par. Caso contrário é ímpar. Também foi feita uma mudança no nome das variáveis para facilitar a compreensão do código.

A propriedade pode ser tão complexa quanto se queira, mas é importante observar que o código de base não muda muito. Observe que há uma leitura, o teste, algum processamento e, por último, a leitura do próximo valor, repetindo-se o código.

Problema: Ler números do teclado, até ler um zero, e imprimir apenas os que são ao mesmo tempo múltiplos de 7 mas não são múltiplos de 2.

Solução similar (apresentada na figura 6.11). Um laço controla o término da repetição, e um desvio condicional verifica a propriedade e algum código é executado em seguida (no caso uma simples impressão na tela).

```
program somapareseimpares;
var x, somapares, somaimpares: integer;

begin
        somapares:= 0;
        somaimpares:= 0;
        read (x);
        while x <> 0 do
        begin
              if x mod 2 = 0 then (* verdadeiro quando x eh par *)
                    somapares:= somapares + x
              else
                    somaimpares:= somaimpares + x;
              read (x);
        end;
        writeln (somapares, somaimpares);
end.
```

Figura 6.10: Soma pares e ímpares.

```
program mult7naopar;
var a: integer;

begin
        read (a);
        while a <> 0 do
        begin
              if (a mod 7 = 0) AND (a mod 2 <> 0) then
                    writeln (a);
              read (a);
        end;
end.
```

Figura 6.11: Imprime os múltiplos de 7 que não são múltiplos de 2.

Um terceiro problema similar é apresentado a seguir.

Problema: Ler números do teclado, até ler um zero, e imprimir apenas os que forem múltiplos de 3 maiores do que 50 e menores ou iguais a 201.

Novamente, mesmo raciocínio (solução na figura 6.12). Apenas a propriedade a ser satisfeita é mais complexa, envolve uma expressão booleana contendo conectivos de conjunção. É importante para o estudante comparar os três últimos códigos e perceber a enorme similaridade entre eles.

6.3.3 Convertendo para binário

Este problema é um desafio que pode ser resolvido com a mesma ideia de se aninhar um desvio condicional sob o escopo de uma repetição, combinada com o uso dos

```
program mult3entre51e201;
var a: integer;

begin
        read (a);
        while a <> 0 do
        begin
             if (a mod 3 = 0) AND (a > 50) AND (a <= 201) then
                  writeln (a);
             read (a);
        end;
end.
```

Figura 6.12: Imprime os múltiplos de 3 lidos entre 51 e 201.

acumuladores.

Problema: Dado um número inteiro entre 0 e 255 imprimir este número em seu formato binário.

Este é um exemplo clássico em que o estudante esquece da teoria. Tradicionalmente, o professor apresenta a definição de números binários, como sendo uma série especial de potências de 2. Em seguida apresenta o "algoritmo para conversão para binário", baseado em uma sequência de divisões por 2. O resultado se obtém tomando-se os restos das sucessivas divisões por 2 "ao contrário", isto é, do último resto de divisão por 2 até o primeiro. Por exemplo, tomando o decimal 22 como entrada:

```
22 div 2
 0     11 div 2
        1     5 div 2
              1     2 div 2
                    0     1 div 2
                          1     0
```

Então o número binário correspondente ao 22 seria 10110. O programa ilustrado na figura 6.13 mostra o código para esta versão do algoritmo.

Ocorre que este algoritmo imprime o binário ao contrário, isto é, a saída para a entrada 22 seria 01101 e não 10110 como deveria. Na verdade, basta lembrar que desde o início deste texto se insiste em afirmar que para um mesmo problema existem diversas soluções. Neste caso, basta usar a definição de números binários.[1]

De fato, o número decimal 22 pode ser escrito numa série de potências de 2 como segue:

$$22 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

[1]Este exemplo nos ocorreu após trocas de emails com o Allan Neves, então aluno da disciplina. Agradecemos a ele por isto.

```
program converteparabinario;
const max=128;
var n: integer;

begin
     write ('entre com um numero entre 0 e 255: ');
     read (n);
     while n <> 0 do
     begin
          write (n mod 2);
          n:= n div 2;
     end;
end.
```

Figura 6.13: Convertendo para binário, versão 1.

A questão é saber quantas vezes cada potência de 2 cabe no número original. Este cálculo é simples e a solução é mostrada na figura 6.14.

```
program converteparabinario_v2;
const max=128;
var diferenca, n, pot2: integer;

begin
     write ('entre com um numero entre 0 e 255: ');
     read (n);
     pot2:= max;
     while pot2 <> 0 do
     begin
          diferenca:= n - pot2;
          if diferenca >= 0 then
          begin
               write (1);
               n:= diferenca;
          end
          else
               write (0);
          pot2:= pot2 div 2;
     end;
end.
```

Figura 6.14: Convertendo para binário, versão 2.

6.3.4 Menor de 3, segunda versão

Generalizando o problema de imprimir o menor entre três números de entrada, vamos agora ler uma sequência de trincas de números e imprimir, para cada entrada, o menor deles. A entrada termina quando os três números lidos forem iguais. Veja como fica o programa na figura 6.15.

```
program imprime_menor_v2 (input,utput);
var
    a, b, c: Integer;

begin
     write('Entre com tres numeros inteiros: ');
     read(a, b, c);
     while not ((a = b) and (b = c)) do (* falso quando a=b=c *)
     begin
          if a < b then
               if a < c then
                    writeln ('o menor dos tres eh ',a)
               else
                    writeln ('o menor dos tres eh ',c)
          else
               if a < c then
                    writeln ('o menor dos tres eh ',b)
               else
                    writeln ('o menor dos tres eh ',c)

          write('entre com tres numeros inteiros: ');
          read(a, b, c);
     end;
end.
```

Figura 6.15: Imprimir o menor dentre 3 números lidos.

6.4 Repetições dentro de condições

Os problemas da seção anterior envolveram a mesma estrutura básica, isto é, um teste sob controle do comando de repetição, com ou sem a parte *else* do *if*. Agora veremos o contrário, isto é, um comando de repetição no escopo do comando de desvio condicional.

6.4.1 Calculo do MDC

Problema: Imprimir o Máximo Divisor Comum (MDC) entre dois números dados.

O conceito matemático de máximo divisor comum entre dois números dados a e b envolve a fatoração de cada número como um produto de fatores primos, escolhendo-se os fatores primos que se repetem com a potência mínima.

Exemplo: Calcular o MDC entre 72 e 135.

$$72 = 2^3 \times 3^2$$
$$135 = 3^3 \times 5$$

Da teoria conclui-se que o MDC entre 72 e 135 é 3^2, pois o 3 é o único fator primo que se repete em ambos os números de entrada, e a menor potência comum é 2.

Implementar este algoritmo para encontrar o MDC é complicado no momento pois não sabemos ainda como obter uma sequência de primos. Também não sabemos ainda o quanto caro é calcular um número primo.

Euclides propôs um algoritmo eficiente para se obter o MDC entre dois números que não requer o uso da fatoração. Trata-se de um dos primeiros algoritmos conhecidos, pois foi proposto por volta do ano 300 a.c. O algoritmo pode ser visto atualmente em *Pascal* conforme está ilustrado na figura 6.16.

```
program mdcporeuclides;
var a, b, resto: integer;

begin
        read (a,b);
        if (a <> 0) AND (b <> 0)) then
        begin
              resto:= a mod b;
              while resto <> 0 do
              begin
                    a:= b;
                    b:= resto;
                    resto:= a mod b;
              end;
              writeln ('mdc = ', b);
        end
        else
              writeln ('o algoritmo nao funciona para entradas nulas.');
end.
```

Figura 6.16: Algoritmo de Euclides para cálculo do MDC.

Este algoritmo mostra um caso em que o comando de repetição está dentro do escopo do comando de desvio condicional. Efetivamente os cálculos são feitos somente se as entradas não forem nulas.

6.5 Repetições aninhadas

Nesta seção estudaremos problemas para os quais os algoritmos exigem o aninhamento de repetições.

6.5.1 Tabuada

Problema: Imprimir as tabuadas do 1 ao 10.

```
program tabuada;
var i,j: integer;

begin
        i:= 1;
        while i <= 10 do
        begin
              j:= 1;
              while j <= 10 do
              begin
                    writeln (i,'x',j,'= ',i*j);  (* comando mais interno *)
                    j:= j + 1;
              end;
              writeln;
              i:= i + 1;
        end;
end.
```

Figura 6.17: Tabuadas do 1 ao 10.

No código apresentado na figura 6.17, para cada valor de i, os valores de j variam de 1 a 10, o que faz com que o comando `writeln` mais interno seja executado 100 vezes ao longo do programa. Se fôssemos imprimir a tabuada do 1 ao 1000, este comando seria executado um milhão de vezes. Em termos genéricos, para uma entrada de tamanho n, o comando mais interno é executado da ordem de n^2 vezes, por isto são conhecidos como *algoritmos de complexidade quadrática*. Nos programas anteriores, o máximo que tínhamos era uma complexidade *linear* com relação ao tamanho da entrada.

6.5.2 Fatorial

O problema seguinte é bastante relevante neste ponto, pois nos permitirá uma análise sobre a técnica usada para resolvê-lo. A técnica básica implementa diretamente a definição de fatorial e usa uma estrutura baseada em um aninhamento triplo: um desvio condicional no escopo de um aninhamento duplo.

Problema: Imprimir o valor do fatorial de todos os números entre 1 e n, sendo n fornecido pelo usuário.

O fatorial de n é assim definido:

$$n = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

Logo, o cálculo do fatorial de um número tem complexidade linear com relação ao tamanho da entrada e pode ser facilmente implementado usando a técnica elementar dos acumuladores, conforme é ilustrado na figura 6.18.

```
program fatorial;
var i, n, fat: integer;

begin
        read (n);
        fat:= 1; (* inicializacao do acumulador *)
        i:= n;
        while  i >= 1 do
        begin
             fat:= fat * i;
             if i > 1 then
                  write (i,'x')
             else
                  write (i,'= ');
             i:= i - 1;
        end;
        writeln (fat);
end.
```

Figura 6.18: Obtendo o fatorial de n.

Esta versão resolve o problema para o cálculo do fatorial de um único número dado como entrada e portanto não resolve o enunciado. Porém, basta colocar este trecho sob o controle de outra repetição que se obtém o efeito desejado. A solução é apresentada na figura 6.19.

É verdade que o programa funciona, mas é extremamente ineficiente e repleto de cálculos redundantes. De fato, o usuário que testou o programa para o valor de $n = 7$ vai perceber que a multiplicação de 2×1 ocorreu 6 vezes, a multiplicação de 3×2 ocorreu 5 vezes, a de 4×6 ocorreu 4 vezes e assim por diante. O programador não explorou corretamente os cálculos já feitos anteriormente. i

Em outras palavras, o programa pode ficar mais eficiente se for feito como ilustrado na figura 6.20, de maneira que o algoritmo, mais esperto, tem complexidade linear, e não quadrática como na versão anterior. Interessante observar que a solução, eficiente, usa apenas uma atribuição no escopo de uma repetição.

```
program fatorial1_n;
var cont, i, n, fat: integer;

begin
        read (n);
        cont:= 1;
        while cont <= n do
        begin
             fat:= 1; (* inicializacao do acumulador *)
             i:= cont;
             while  i >= 1 do
             begin
                  fat:= fat * i;
                  if i > 1 then
                       write (i,'x')
                  else
                       write (i,'= ');
                  i:= i - 1;
             end;
             writeln (fat);
             cont:= cont + 1;
        end;
end.
```

Figura 6.19: Obtendo vários fatoriais.

```
program fatorial1_n_v2;
var cont, n, fat: integer;

begin
        read (n);
        cont:= 1;
        fat:= 1; (* inicializacao do acumulador *)
        while cont <= n do
        begin
             fat:= fat * cont;
             writeln ('fat(',cont,')= ', fat);
             cont:= cont + 1;
        end;
end.
```

Figura 6.20: Otimizando o cálculo dos fatoriais.

6.6 Exercícios

1. Dados dois números inteiros positivos determinar o valor da maior potência do primeiro que divide o segundo. Se o primeiro não divide o segundo, a maior potência é definida igual a 1. Por exemplo, a maior potência de 3 que divide 45 é 9.

   ```
   Entrada:        Saída:
   3 45            9
   ```

2. Dadas as populações P_A e P_B de duas cidades A e B em 2009, e suas respectivas taxas de crescimento anual X_A e X_B, faça um programa em *Pascal* que receba estas informações como entrada e determine:

   - se a população da cidade de menor população ultrapassará a de maior população;
   - e o ano em que isto ocorrerá.

3. Um inteiro positivo N é perfeito se for igual a soma de seus divisores positivos diferentes de N.

   Exemplo: 6 é perfeito pois $1 + 2 + 3 = 6$ e $1, 2, 3$ são todos os divisores positivos de 6 e que são diferentes de 6.

   Faça um programa em *Pascal* que recebe como entrada um número positivo K e mostre os K primeiros números perfeitos.

4. Faça um programa em *Pascal* que dado um inteiro positivo n, escreva todos os termos, do primeiro ao n-ésimo, da série abaixo. Você pode assumir que o usuário nunca digita valores menores que 1 para n.

   $$5, 6, 11, 12, 17, 18, 23, 24, \ldots$$

5. Faça um programa em *Pascal* que, dada uma sequência de números naturais positivos terminada por 0 (zero), imprimir o histograma da sequência dividido em quatro faixas (o histograma é a contagem do número de elementos em cada faixa):

   - Faixa 1: 1 – 100;
   - Faixa 2: 101 – 250;
   - Faixa 3: 251 – 20000;
   - Faixa 4: acima de 20001.

   Exemplo:

   ```
   Entrada: 347 200 3 32000 400 10 20 25 0
   Saída:  Faixa 1: 4
           Faixa 2: 1
           Faixa 3: 2
           Faixa 4: 1
   ```

6. Fazer um programa em *Pascal* que leia uma sequência de pares de números inteiros quaisquer, sendo dois inteiros por linha de entrada. A entrada de dados termina quando os dois números lidos forem nulos. Este par de zeros não deve ser processado e serve apenas para marcar o término da entrada de dados.

   Para cada par A, B de números lidos, se B for maior do que A, imprimir a sequência $A, A+1, \ldots, B-1, B$. Caso contrário, imprimir a sequência $B, B+1, \ldots, A-1, A$.

   Exemplos:

```
Entrada      Saida
4 6          4 5 6
-2 1         -2 -1 0 1
2 -3         -3 -2 -1 0 1 2
0 0
```

7. Fazer um programa em *Pascal* que receba um número inteiro N como entrada e imprima cinco linhas contendo as seguintes somas, uma em cada linha:

```
N
N + N
N + N + N
N + N + N + N
N + N + N + N + N
```

   Exemplo:

```
                        Saída:
                           3
Entrada:                   6
3                          9
                           12
                           15
```

8. Fazer um programa em *Pascal* que imprima exatamente a saída especificada na figura 1 (abaixo) de maneira que, em todo o programa fonte, não apareçam mais do que três comandos de impressão.

```
1
121
12321
1234321
123454321
12345654321
1234567654321
123456787654321
12345678987654321
```

Figura 1

9. Fazer um programa em *P*ascal que imprima exatamente a mesma saída solicitada no exercício anterior, mas que use exatamente dois comandos de repetição.

10. Adaptar a solução do exercício anterior para que a saída seja exatamente conforme especificada na figura 2 (abaixo).

```
1
121
12321
1234321
123454321
12345654321
1234567654321
123456787654321
12345678987654321
```

Figura 2

11. Enumere e explique todos os erros contidos no seguinte código *Pascal*:

```
program misterio2;
var
  m, g: real;
  N1, N2: integer;
begin
  readln(N1, N2);
  if (N1 > N2) then
    m := N2
  else
    m := N1;
  g:= 1;
  while g do
  begin
    if (N1 mod m = 0) AND (N2 mod m = 0) then
      g := 0;
    else
      m := m - 1;
  end;
  if (m := N1) then
    writeln('O valor resultante eh: ' m);
end.
```

12. Leia do teclado uma sequência de $N > 0$ números quaisquer. Para cada valor lido, se ele for positivo, imprimir os primeiros 10 múltiplos dele.

13. Sabe-se que um número da forma n^3 é igual a soma de n números ímpares consecutivos.

    Exemplos:

    - $1^3 = 1$

- $2^3 = 3 + 5$
- $3^3 = 7 + 9 + 11$
- $4^3 = 13 + 15 + 17 + 19$

Dado M, escreva um program em *Pascal* que determine os ímpares consecutivos cuja soma é igual a n^3 para n assumindo valores de 1 a M.

14. Faça um programa em Pascal que, dados dois números naturais m e n determinar, entre todos os pares de números naturais (x, y) tais que $x <= m$ e $y <= n$, um par para o qual o valor da expressão $xy - x^2 + y$ seja máximo e calcular também esse máximo.

15. (*) Escreva um programa em *Pascal* para ler uma sequência de números inteiros, terminada em -1. Para cada número inteiro lido, o programa deve verificar se este número está na base binária, ou seja, se é composto apenas pelos dígitos 0 e 1. Caso o número esteja na base binária, o programa deve imprimir seu valor na base decimal. Caso contrário, deve imprimir uma mensagem indicando que o número não é binário. Ao final do programa deve ser impresso, em formato decimal, o maior número válido (binário) da sequência.

    Dica: dado o número 10011 em base binária, seu valor correspondente em base decimal será dado por

    $$1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 19$$

    Exemplo:

    ```
    Entrada:    Saida:
    10011       19
    121         numero nao binario
    1010        10
    101010101   341
    0           0
    -1

    O maior numero binario foi 341
    ```

Capítulo 7

Aplicações das técnicas elementares

Neste capítulo vamos escrever soluções para problemas cujo grau de dificuldade é similar aos dos problemas do capítulo anterior, com o objetivo de fixar conceitos. Nestes problemas as técnicas devem ser combinadas com inteligência, pios deve-se pensar em como resolver problemas complexos usando-se apenas os elementos básicos. A partir de agora omitiremos os cabeçalhos dos programas em *Pascal*.

7.1 Números de Fibonacci

Esta é uma das mais famosas sequências de números que existe. Trata-se dos números de Fibonacci[1]. Esta sequência é gerada de modo bastante simples. Os dois primeiros valores são 1 e 1. Os seguintes são obtidos pela soma dos dois anteriores. Assim, a sequência de Fibonacci é: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Problema: Imprimir os primeiros números da sequência de Fibonacci.

Como gerar e imprimir os elementos desta sequência é o nosso desafio. A solução exige que se guarde sempre os dois últimos elementos gerados, senão não é possível resolver o problema. Observamos que a frase do parágrafo anterior dá a dica: "os seguintes são obtidos pela soma dos dois anteriores".

Na solução apresentada na figura 7.1 consideramos duas variáveis importantes, uma para armazenar o último elemento já produzido pelo algoritmo, a outra para guardar o penúltimo. Com estes dois, é possível produzir a soma deles, isto é, o próximo elemento.

A atualização destas variáveis deve ser feita sempre que o próximo elemento for obtido, pois a soma do último com o penúltimo é agora o último elemento gerado. O que era o último passa a ser o penúltimo. A ordem da atualização destes valores é relevante no código em função do esquema de funcionamento de memória do computador. Trocando-se a ordem dos comandos o algoritmo para de funcionar.

O algoritmo da figura 7.1 é normalmente o mais utilizado para este problema, isto é, define-se os dois primeiros e depois sucessivamente soma-se os dois últimos e

[1]Vale a pena ler: http://pt.wikipedia.org/wiki/Número_de_Fibonacci.

```
program Fibonacci;
const max=93;  (* A partir do 94o estoura a capacidade do tipo qword *)
var ultimo, penultimo, soma, cont: integer;
begin
      ultimo:= 1;            (* inicializacao das variaveis principais *)
      penultimo:= 1;
      writeln ('1 ',penultimo);  (* imprime os dois primeiros valores *)
      writeln ('2 ',ultimo);
      cont:= 3 ;                  (* calcula do terceiro em diante *)
      while cont <= max do
      begin
            soma:= penultimo + ultimo;
            writeln (cont, ' ',soma);
            penultimo:= ultimo;        (* a ordem destes dois comandos *)
            ultimo:= soma;             (* eh relevante no codigo       *)
            cont:= cont + 1;
      end;
end.
```

Figura 7.1: Gerando números da sequência de Fibonacci.

atualiza-se o último como sendo esta soma recentemente gerada e o penúltimo como sendo o que era o último. Existem vários outros que são apresentados como exercícios.

Um problema similar mas alternativo a este é, por exemplo, saber qual é o primeiro número de Fibonacci maior do que um determinado valor. Uma pequena alteração no controle de parada e no local do comando de impressão resolve o novo problema. Isto é apresentado na figura 7.2. A diferença básica é que neste caso não é preciso contar os números, pois o critério de parada é diferente. Mas é importante observar que a parte da geração dos números da sequência não mudou.

```
program Fibonacci_2;
const max=1000;
var ultimo, penultimo, soma, cont: integer;
begin
      ultimo:= 1;            (* inicializacao das variaveis principais *)
      penultimo:= 1;
      soma:= penultimo + ultimo;
      while soma <= max do          (* calcula do terceiro em diante *)
      begin
            penultimo:= ultimo;
            ultimo:= soma;
            soma:= penultimo + ultimo;
      end;
      writeln (soma);
end.
```

Figura 7.2: Imprimindo o primeiro número de Fibonacci maior do que 1000.

Uma das maiores belezas dos números de Fibonacci é que a razão entre dois

termos consecutivos converge para um número irracional conhecido como *número áureo* (também podendo ter outros nomes parecidos). Também é denotado pela letra grega φ e é aproximadamente 1.6180339887499.

De fato, vejamos a razão entre dois termos consecutivos para alguns números pequenos:

$$\frac{1}{1} = 1, \frac{2}{1} = 2, \frac{3}{2} = 1.5, \frac{5}{3} = 1.66, \frac{8}{5} = 1.60, \frac{13}{8} = 1.625, \frac{21}{13} = 1.615, \ldots$$

O algoritmo que calcula o número áureo com a precisão desejada, mostrando os valores intermediários, é apresentado na figura 7.3. Notamos que ele verifica a convergência da sequência para a razão áurea, fazendo as contas até que o erro seja menor que a precisão desejada. A função *abs* é nativa do *Pascal* e retorna o valor absoluto do número dado como argumento.

Neste exemplo, novamente o cálculo central não mudou, isto é, os números continuam a ser gerados da mesma maneira. O que muda *é o que fazer com eles*. No caso, é preciso obter a razão do último pelo penúltimo. Também é o primeiro exemplo apresentado que usa números reais ao invés de inteiros.

```
program numero_aureo;
const PRECISAO=0.00000000000001;
var ultimo, penultimo, soma: integer;
    naureo, naureo_anterior: real;
begin
     ultimo:= 1;       (* inicializacao das variaveis principais *)
     penultimo:= 1;
     naureo_anterior:= -1;  (* para funcionar o primeiro teste  *)
     naureo:= 1;
     writeln (naureo:15:14);
                              (* calcula do terceiro em diante *)
     while abs(naureo - naureo_anterior) >= PRECISAO do
     begin
          soma:= penultimo + ultimo;
          naureo_anterior:= naureo;
          naureo:= soma/ultimo;
          writeln (naureo:15:14);
          penultimo:= ultimo;
          ultimo:= soma;
     end;
end.
```

Figura 7.3: Verificando a convergência do número áureo.

Várias outras propriedades interessantes serão deixadas como exercício.

7.2 Maior segmento crescente

Problema: Dada uma sequência de n números naturais, imprimir o valor do comprimento do segmento crescente de tamanho máximo dentre os números lidos.

- Por exemplo, se a sequência for: 5, 10, 3, 2, 4, 7, 9, 8, 5, o comprimento crescente máximo é 4, pois é o tamanho do segmento 2, 4, 7, 9.
- Na sequência 10, 8, 7, 5, 2, o comprimento de um segmento crescente máximo é 1.

O algoritmo que resolve este problema é apresentado na figura 7.4. A solução exige um conjunto de variáveis que controlam o estado da computação, isto é, que tentam manter uma certa memória de que tipo de números foi lido até o momento, segundo uma dada restrição. Os comentários no código ilustram como guardar o tamanho da sequência sendo lida no momento em comparação com o maior tamanho lido até o momento.

Em um certo sentido guarda a mesma ideia do algoritmo para a sequência de Fibonacci, pois é necessário guardar o valor lido anteriormente para possibilitar a comparação com o valor atual. Isto é necessário para saber se o valor atual é maior que o anterior. Se for maior, estamos em uma sequência crescente. Caso contrário, trata-se de outra sequência. A variável *tamanho* armazena o tamanho da sequência crescente sendo lida, enquanto que *maiortam* registra a maior sequência crescente que já foi processada até o momento.

7.3 Séries

Nesta seção tratamos de problemas que envolvem o cálculo de séries, normalmente utilizadas para cálculos de funções tais como seno, cosseno, logaritmo, etc... A técnica utilizada é basicamente aquela dos acumuladores, porém, o cálculo dos novos termos somados é ligeiramente mais sofisticado e exige atenção do estudante para a boa compreensão da solução.

7.3.1 Número neperiano

Problema: Calcular o valor no número $e = 2.718281\ldots$ pela série abaixo, considerando apenas os vinte primeiros termos:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} + \ldots$$

```
program maior_seg_crescente;
var n, tamanho, maiortam, a_anterior, i, a: integer;
begin
     read (n);                    (* inicializa as variaveis principais *)
     tamanho:= 0;
     maiortam:= 0;
     a_anterior:= 0;              (* zero eh o primeiro numero natural  *)

     i:= 1;
     while i <= n do
     begin
          read (a);
          if a > a_anterior then
               tamanho:= tamanho + 1
          else
          begin
               if tamanho > maiortam then (* lembra o maior tamanho *)
                    maiortam:= tamanho;
               tamanho:= 1;                (* reinicializa o contador *)
          end;
          a_anterior:= a;                (* lembra do numero anterior *)
          i:= i + 1;
     end;
       (* este ultimo if testa se a ultima sequencia nao eh a maior *)
     if tamanho > maiortam then           (* lembra o maior tamanho *)
          maiortam:= tamanho;
     writeln ('maior tamanho encontrado: ',maiortam);
end.
```

Figura 7.4: Maior segmento crescente.

A figura 7.5 ilustra uma solução baseada nas já estudadas técnicas dos acumuladores e no problema dos fatoriais. Basicamente, o novo termo é calculado em função do fatorial, que, neste caso, aparece no denominador.

O programa poderia ter dispensado o uso da variável *novotermo*, mas deixamos assim pois facilita a compreensão de que a grande e nova dificuldade neste problema é "como gerar o novo termo a partir do anterior"? Isto vai ajudar na solução de problemas similares, na sequência do texto.

Neste caso, o enunciado do problema determinou que o programa calculasse 20 termos da série. Alguns enunciados estabelecem como condição de parada um critério de erro, isto é, se o cálculo em uma iteração difere do cálculo anterior por um valor previamente estabelecido, isto é, uma precisão previamente determinada. Em alguns casos, quando o cálculo da série não é convergente, este valor mínimo nunca é atingido, obrigando também o programa a testar um número máximo de iterações. Segue a mesma ideia discutida no caso de gerar o número áureo.

Suponhamos que o enunciado tivesse como condição de parada um destes dois critérios: ou o erro é menor do que 10^{-4} ou o número máximo de iterações é 50. Se o programa sair pelo critério de erro, então o valor de e foi encontrado, senão, se saiu pelo critério do número máximo de iterações, então o programa avisa que

```
program neperiano;
var e, novotermo: real;
    fat, i: longint;
const itmax=20;
begin
      e:= 0;                          (* inicializacao das variaveis *)
      fat:= 1;
      i:= 1;                                        (* calculo da serie *)
      while i <= itmax do
      begin
            novotermo:= 1/fat;
            e:= e + novotermo;
            fat:= i*fat;
            i:= i + 1;
      end;
      writeln ('e= ',e);
end.
```

Figura 7.5: Cálculo do número neperiano.

não conseguiu encontrar a solução. A figura 7.6 ilustra a solução para este caso. É importante observar que o que muda com relação à primeira solução é apenas o critério de parada. Os cálculos internos são os mesmos.

7.3.2 Cálculo do seno

Problema: Calcular o valor de $seno(x)$ pela série abaixo, considerando apenas os vinte primeiros termos:

$$seno(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \ldots$$

Comparando esta série com a do número neperiano, observamos três diferenças básicas: a primeira é o sinal, que a cada termo muda de positivo para negativo e vice-e-versa e antes era sempre positivo; a segunda é o numerador, que antes era a constante 1, agora é uma potência de x; a terceira é que os fatoriais não são consecutivos em cada termo, aumentam de dois em dois.

Trataremos de cada caso separadamente, construindo a solução para o seno baseada na do número neperiano. Vamos tratar primeiro o mais simples, que é a questão dos sinais. Considere a seguinte série:

$$tmp = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \frac{1}{5!} + \frac{1}{6!} - \frac{1}{7!} + \ldots$$

```
program neperiano_v2;
const itmax=50; precisao=0.0001;
var e, eanterior, novotermo: real;
    fat, i: integer;
begin
      e:= 0;
      eanterior:= -1;
      fat:= 1;
      i:= 1;
      while (i <= itmax) and (e - eanterior > precisao) do
      begin
           novotermo:= 1/fat;
           eanterior:= e;
           e:= e + novotermo;
           fat:= i*fat;
           i:= i + 1;
      end;
      if i > itmax then
           writeln ('solucao nao encontrada')
      else
           writeln ('e= ',e);
end.
```

Figura 7.6: Cálculo do número neperiano.

O algoritmo da figura 7.7 modifica o anterior pela introdução de uma nova variável que controla a mudança do sinal, produzindo como saída o cálculo da função *tmp*.

Agora vamos corrigir o denominador, obtendo a seguinte série:

$$tmp2 = \frac{1}{1!} - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \frac{1}{9!} - \frac{1}{11!} + \frac{1}{13!} - \frac{1}{15!} + \ldots$$

A atualização da variável fat deve ser modificada, conforme apontado na figura 7.8:

Agora só resta a atualização correta do numerador, que depende da leitura de um valor x do teclado. Uma variável adicional contém o valor de x^2, para evitar cálculos redundantes. O programa ilustrado na figura 7.9 implementa corretamente o cálculo do $seno(x)$ para um x dado em radianos.

Existem várias outras séries que podem ser implementadas com estas mesmas ideias, vamos deixar isto como exercício. Basicamente, basta o estudante observar como os numeradores e denominadores podem ser atualizados em função do termo anteriormente calculado.

```
program serie_v1;
const itmax=20;
var seno, novotermo: real;
    fat, sinal, i: integer;
begin
      seno:= 0;
      fat:= 1;
      sinal:= 1;
      i:= 1;
      while i <= itmax do
      begin
           novotermo:= 1/fat;
           seno:= seno + sinal*novotermo;
           sinal:= -sinal;        (* nova variavel para sinal trocado *)
           fat:= i*fat;
           i:= i + 1;
      end;
      writeln ('tmp= ',seno);
end.
```

Figura 7.7: Série com troca de sinais.

```
program serie_v2;
const itmax=20;
var seno, novotermo: real;
    fat, sinal, i: integer;
begin
      seno:= 0;
      fat:= 1;
      sinal:= 1;
      i:= 1;
      while i <= itmax do
      begin
           novotermo:= 1/fat;
           seno:= seno + sinal*novotermo;
           fat:= 2*i*(2*i+1)*fat;        (* esta eh a princial modificacao *)
           sinal:= -sinal;
           i:= i + 1;
      end;
      writeln ('tmp2= ',seno);
end.
```

Figura 7.8: Série com troca de sinais e fatorais no denominador corrigidos.

```
program senox;
const itmax=20;
var seno, x, x_quadrado, numerador, novotermo: real;
    fat, i, sinal: integer;
begin
      seno:= 0;
      fat:= 1;
      sinal:= 1;
      read (x);
      x_quadrado:= x*x;
      numerador:= x;
      i:= 1;
      while i <= itmax do
      begin
            novotermo:= numerador/fat;              (* atualiza o termo *)
            seno:= seno + sinal*novotermo;
            numerador:= numerador*x_quadrado;   (* atualiza o numerador *)
            fat:= 2*i*(2*i+1)*fat;
            sinal:= -sinal;
            i:= i + 1;
      end;
      writeln ('seno(',x,')= ',seno);
end.
```

Figura 7.9: Cálculo do seno de x.

7.4 Números primos

Este é um dos mais completos problemas desta parte inicial da disciplina. A solução dele envolve diversos conceitos aprendidos e, de certa forma, permite o estabelecimento de um ponto de checagem no aprendizado.

Problema: Dado um número natural n, determinar se ele é primo.

Números naturais primos são aqueles que são divisíveis apenas por ele mesmo e por 1. Em outras palavras, se n é um número natural, então ele é primo se, e somente se, não existe outro número $1 < p < n$ que divida n.

Os números primos são de particular interesse em computação, sobretudo nos dias de hoje, pois estão por trás dos principais algoritmos de criptografia e transmissão segura na Internet. Nesta seção vamos estudar alguns algoritmos para se determinar se um número é ou não primo.

Aplicando-se diretamente a definição, temos que verificar se algum número entre 2 e $n-1$ divide n. Se p divide n então o resultado da operação $nmod\ p$ é zero.

O primeiro algoritmo, apresentado na figura 7.10, é bastante simples: basta variar p de 2 até $n-1$ e contar todos os valores para os quais p divide n. Se a contagem for zero, o número não tem divisores no intervalo e é portanto primo. Senão não é.

```
program ehprimo;
var n, cont, i: integer;
begin
      read (n);
      cont:= 0; (* contador de divisores de n *)
      i:= 2;
      while i <= n-1 do
      begin
            if n mod i = 0 then
                  cont:= cont + 1; (* achou um divisor *)
            i:= i + 1;
      end;
      if cont = 0 then
            writeln (n,' eh primo')
end.
```

Figura 7.10: Verifica se n é primo contando os divisores.

Este algoritmo é bom para se determinar *quantos* divisores primos um dado número tem, mas não é eficiente para este problema pois basta saber se *existe pelo menos um* divisor. Logo, basta parar logo após o primeiro divisor ter sido encontrado.

A técnica utilizada é baseada em uma variável booleana inicializada como sendo verdadeira. O algoritmo "chuta" que n é primo mas, em seguida, se os cálculos mostrarem que o chute estava errado, a informação é corrigida.

O laço principal do programa deve ter duas condições de parada: (1) termina quando um divisor foi encontrado; (2) termina quando nenhum divisor foi encontrado,

isto é, quando i ultrapassou $n-1$. Um teste na saída do laço encontra o motivo da saída e imprime a resposta correta. Este algoritmo pode ser visto na figura 7.11.

```
program ehprimo_v2;
var n, i: integer;
    eh_primo: boolean;
begin
     read (n);
     eh_primo:= true; (* inicia chutando que n eh primo *)
     i:= 2;
     while (i <= n-1) and eh_primo do
     begin
          if n mod i = 0 then
               eh_primo:= false; (* se nao for, corrige *)
          i:= i + 1;
     end;
     if eh_primo then
          writeln (n,' eh primo')
end.
```

Figura 7.11: Testa se n é primo parando no primeiro divisor.

A análise deste algoritmo deve ser feita em duas situações: (1) no pior caso, aquele em que o número é primo; (2) no melhor caso, aquele em que o número não é primo, de preferência um primo bem baixo.

No segundo caso, o algoritmo vai terminar bem rápido. No outro, ele vai testar todas as possibilidades de ímpares. Mas o caso ótimo é raro. De fato, nos problemas envolvendo criptografia, estes números primos tem duzentos ou mais dígitos. Isto pode fazer com que o computador fique bastante tempo processando a informação.

Percebemos então que se o número for par então só tem uma chance dele ser também primo, justamente se o número for o 2. No caso dos ímpares é necessário que todos sejam testados. A figura 7.12 ilustra o programa que implementa este raciocínio.

O algoritmo testa inicialmente se o número é par. Se for, testa se é o 2, que é primo. Se for ímpar, testa todos os ímpares entre 1 e $n-1$, eliminando metade dos cálculos, pois os pares foram descartados.

Melhorou, mas pode melhorar mais com um pouco mais de observação: não é necessário se testar *todos* os ímpares, basta que se teste até a *raiz* no número.

De fato, todo número natural pode ser decomposto como um produto de números primos. Se a entrada não for um primo, então pode ser decomposta, no melhor caso, assim: $n = p * p$, em que p é primo. O algoritmo que implementa esta solução é mostrado na figura 7.13.

Para se ter uma ideia do ganho, vejam na tabela seguinte o quanto se ganha com as três últimas versões do programa.

x	$\frac{x}{2}$	$\sqrt{x}$
1000000	500000	1000
1000000000	500000000	31622
1000000000000	500000000000	1000000

```
program eh_primo_v3;
var n, i: integer;
    eh_primo: boolean;
begin
     read (n);
     eh_primo:= true; (* inicia chutando que n eh primo *)
     if n mod 2 = 0 then (* n eh par *)
          if n <> 2 then
               eh_primo:= false (* n nao eh 2 *)
          else eh_primo:= true
     else (* n nao eh par, testar todos os impares *)
     begin
          i:= 3;
          while (i <= n-1) and eh_primo do
          begin
               if n mod i = 0 then
                    eh_primo:= false; (* achamos um divisor impar *)
               i:= i + 2;
          end;
     end;
     if eh_primo then
          writeln (n,' eh primo')
end.
```

Figura 7.12: Testa se n é primo, tratando os pares em separado.

A tabela acima mostra que para entradas da ordem de 10^{12}, o número de cálculos feitos com o programa da figura 7.11 pode ser da mesma ordem de 10^{12}. Os cálculos do programa da figura 7.12 pode ser da ordem de 10^{11} (ganho pequeno), enquanto que na última versão, ele pode fazer cálculos da ordem de "apenas" 10^6.

```
program eh_primo_v3;
var n, i: integer;
    eh_primo: boolean;
begin
     read (n);
     eh_primo:= true;              (* inicia chutando que n eh primo *)
     if n mod 2 = 0 then           (* n eh par *)
          if n <> 2 then
               eh_primo:= false    (* n nao eh 2 *)
          else eh_primo:= true
     else begin (* n nao eh par, testar todos os impares *)
          i:= 3;
          while (i <= trunc(sqrt(n))) and eh_primo do
          begin
               if n mod i = 0 then
                    eh_primo:= false; (* achamos um divisor impar *)
               i:= i + 2;
          end;
     end;
     if eh_primo then
          writeln (n,' eh primo')
end.
```

Figura 7.13: Testa se n é primo parando na raiz de n.

7.5 Exercícios

1. Dado um número de três dígitos, construir outro número de quatro dígitos com a seguinte regra: a) os três primeiros dígitos, contados da esquerda para a direita, são iguais aos do número dado; b) o quarto dígito e' um dígito de controle calculado da seguinte forma: primeiro dígito + 3*segundo dígito + 5*terceiro dígito; o dígito de controle é igual ao resto da divisão dessa soma por 7.

2. Dado um número inteiro de cinco dígitos representando um número binário, determinar seu valor equivalente em decimal. Por exemplo, se a entrada for 10001, a saída deve ser 17.

3. Considere o programa feito para resolução do cálculo do número neperiano (seção 7.3.1. Quantas operações de multiplicação sao executadas no seu programa?

   (a) Considerando 20 termos.

   (b) Considerando N termos.

4. Considere a progressão geométrica 1, 2, 4, 8, 16, 32, ... e um inteiro positivo N. Imprimir os N primeiros termos desta PG e a soma deles.

5. Imprimir os N primeiros termos das sequências definidas pelas relações de recorrência:

   (a) $Y(k+1) = Y(k) + k, k = 1, 2, 3..., Y(1) = 1$

   (b) $Y(k+1) = Y(k) + (2k+1), k = 0, 1, 2, 3..., Y(0) = 1$

   (c) $Y(k+1) = Y(k) + (3k^2 + 3k + 1), k = 0, 1, 2, 3..., Y(0) = 1$

   (d) $Y(k+1) = 2Y(k), k = 1, 2, 3..., Y(1) = 1$

6. Dado um número inteiro N, tabelar $N[k]$ para k variando de 1 até N. Considere que, por definição, $X[k] = X(X-1)(X-2)(X-3)...(X-k+1)$, X sendo um número real, k um natural diferente de zero e $X[0] = 1$. Observe que se $X = N = k$, então $N[N] = N!$.

7. Sabe-se que o valor do coseno de 1 (um) radiano pode ser calculado pela série infinita abaixo:

$$coseno(x) = \frac{1}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

   Fazer um programa que calcule o valor do cosseno de x obtido pela série acima considerando apenas os primeiros 14 termos da mesma.

8. Considere o conjunto C de todos os números inteiros com quatro algarismos distintos, ordenados segundo seus valores, em ordem crescente:

$$C = \{1023, 1024, 1025, 1026, 1027, 1028, 1029, 1032, 1034, 1035, \dots\}$$

Faça um programa em *Pascal* que leia um número N, pertencente a este conjunto, e imprima a posição deste número no conjunto.

Exemplos:

- Entrada: 1026
  Saída: 4
- Entrada: 1034
  Saída: 9
- Entrada: 9876
  Saída: 4536
- Entrada: 1243
  Saída: 72

9. Faça um programa em *Pascal* que calcule e imprima o valor de $f(x)$, onde $x \in \Re$ é lido no teclado e:

$$f(x) = \frac{5x}{2!} - \frac{6x^2}{3!} + \frac{11x^3}{4!} - \frac{12x^4}{5!} + \frac{17x^5}{6!} - \frac{18x^6}{7!} + \dots$$

O cálculo deve parar quando $abs(f(x_{n+1}) - f(x_n)) < 0.00000001$, onde $abs(x)$ é a função em *Pascal* que retorna o valor absoluto de x.

10. O número áureo φ (1,6180339...) pode ser calculado através de expressões com séries de frações sucessivas do tipo:

$$\begin{aligned}
\varphi_1 &= 1 + \frac{1}{1} = 2 \\
\varphi_2 &= 1 + \frac{1}{1 + \frac{1}{1}} = 1,5 \\
\varphi_3 &= 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = 1,666 \\
\varphi_4 &= 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}} = 1,6
\end{aligned}$$

onde φ_i indica a aproximação do número áureo com i frações sucessivas. Estes valores variam em torno do número áureo, sendo maior ou menor alternadamente, mas sempre se aproximando deste quando o número de frações cresce.

Faça um programa em *Pascal* que leia um número N e imprima o valor da aproximação do número áureo φ_N, que usa uma série de N frações sucessivas.

11. Dado um inteiro positivo N e dada uma sequência de N números reais $x_1, \dots, x_n$ faça um programa em *Pascal* que calcule o quociente da soma dos reais pelo seu produto. Isto é:

$$q = \frac{\sum_{i=1}^{N} x_i}{\prod_{i=1}^{N} x_i}$$

Como não pode haver divisão por zero, seu programa deve parar tão logo esta situação seja verificada indicando uma mensagem apropriada para o usuário.

12. Em *Pascal* o tipo *CHAR* é enumerável, e portanto está na classe dos tipos chamados de *ordinais*, conforme o guia de referência da linguagem estudado em aula. A ordem de cada caracter é dada pela tabela ASCII. Assim é possível, por exemplo, escrever trechos de códico tais como:

```
IF 'A' > 'B' THEN
     WRITE ('A eh maior que B')
ELSE
     WRITE ('A não eh maior que B');
```

que produziria a mensagem "A não eh maior que B", pois na tabela ASCII o símbolo "A" tem ordem 64 enquanto que "B" tem ordem 65.

Ou ainda:

```
FOR i:= 'a' TO 'z' DO
     WRITE (i);
```

que produziria como saída "abcdefghijklmnopqrstuvxwyz".

Faça um programa em *Pascal* que leia seu nome completo (nomes completos em geral) constituídos por apenas letras maiúsculas entre "A" e "Z" e espaços em branco terminadas em "." e que retorne o número de vogais e consoantes neste nome. Exemplos:

```
Entrada: FABIANO SILVA.
Saída:
     Vogais: 6
     Consoantes: 6

Entrada: MARCOS ALEXANDRE CASTILHO.
Saída:
     Vogais: 9
     Consoantes: 14
```

13. Faça um programa em *Pascal* que leia um inteiro positivo n, e escreva a soma dos n primeiros termos da série:

$$\frac{1000}{1} - \frac{997}{2} + \frac{994}{3} - \frac{991}{4} + \dots$$

14. Dizemos que uma sequência de inteiros é k**-alternante** se for composta alternadamente por segmentos de números pares de tamanho k e segmentos de números ímpares de tamanho k.

    Exemplos:

    A sequência 1 3 6 8 9 11 2 4 1 7 6 8 é 2-alternante.
    A sequência 2 1 4 7 8 9 12 é 1-alternante.
    A sequência 1 3 5 é 3-alternante.

    Escreva um programa *Pascal* que verifica se uma sequência de tamanho n é 10-alternante. O programa deve ler n, o tamanho da sequência, no inicío do programa e aceitar somente valores múltiplos de 10. A saída do programa deve ser a mensagem "`A sequencia eh 10-alternante`" caso a sequência seja 10-alternante e "`A sequencia nao eh 10-alternante`", caso contrário.

15. Faça um programa em *Pascal* que calcule o resultado da seguinte série:

$$S = \frac{x^0}{2!} - \frac{x^4}{6!} + \frac{x^8}{10!} - \frac{x^{12}}{14!} + \frac{x^{16}}{18!} - \cdots$$

16. Faça um programa em *Pascal* que receba como entrada um dado inteiro N e o imprima como um produto de primos. Exemplos: $45 = 3 \times 3 \times 5$. $56 = 2 \times 2 \times 2 \times 7$.

17. Faça um programa em *Pascal* que calcule e escreva o valor de S assim definido:

$$S = \frac{1}{1!} - \frac{2}{2!} + \frac{4}{3!} - \frac{8}{2!} + \frac{16}{1!} - \frac{32}{2!} + \frac{64}{3!} - \cdots$$

18. Fazer um programa em *Pascal* que calcule e escreva o valor de S:

$$S = \frac{37 \times 38}{1} + \frac{36 \times 37}{2} + \frac{35 \times 36}{3} + \cdots + \frac{1 \times 2}{37}$$

Capítulo 8

Refinamentos sucessivos

Neste ponto consideramos que o estudante já domina as estruturas elementares de controle de fluxo de execução de um programa, então podemos aplicar estes conceitos em alguns algoritmos mais sofisticados, isto é, em problemas para os quais as soluções envolvem generalizações dos conteúdos previamente ministrados.

A partir de agora, vamos construir os programas passo a passo, iniciando pela apresentação de um *pseudo-código*, que é um algoritmo de mais alto nível que normalmente não pode ser compilado, mas pode progressivamente ser detalhado até se obter o código final compilável.

Vamos também introduzir uma técnica de se isolar os subproblemas, de maneira a definir blocos que realizam cálculos bem definidos: integrando-se os diversos pedaços, teremos a solução global do problema. Também não nos preocuparemos mais com os cabeçalhos dos programas.

8.1 Primos entre si

Problema: Imprimir todos os pares de números (a, b) que são primos entre si para todo $2 \leq a \leq 100$ e $a \leq b \leq 100$.

Este problema pode ser dividido em dois sub-problemas: (1) dado um par de números quaisquer, como saber se eles são primos entre si? (2) como gerar todos os pares ordenados no intervalo desejado?

Nosso conhecimento deveria ser suficiente para resolvermos o segundo sub-problema. Por este motivo vamos iniciar a tentativa de solução por ele, neste ponto ignorando completamente o primeiro sub-problema.

A solução poderia ser tal como mostrado na figura 8.1, a qual apresenta um código global que gera todos os pares ordenados que interessam, isto é, tais que $2 \leq a \leq 100$ e $a \leq b \leq 100$.

Na parte central do pseudo-código existe um desvio condicional que não está escrito em forma de linguagem de programação, mas que pode ser considerado como um teste feito através de um *oráculo* que, por sua vez, sabe responder sim ou não se cada par é ou não é constituído por pares de números primo entre si.

```
begin
      i:= 2;
      while i <= 100 do
      begin
            j:= i; (* para gerar pares com j sendo maior ou igual a i *)
            while  j <= 100 do
            begin
                  if {i e j sao primos entre si} then writeln (i,j); (* oraculo *)
                  j:= j + 1;
            end;
            i:= i + 1;
      end;
end.
```

Figura 8.1: Pseudo-código para o problema dos primos entre si.

Agora basta transformar o oráculo em código propriamente dito. Lembramos que a e b são números primos entre si quando o máximo divisor comum entre eles é 1. Isto é, $mdc(a, b) = 1$. Na seção 6.4.1 vimos como se calcula de forma eficiente o MDC entre dois números pelo método de Euclides (figura 6.16. A versão final está na figura 8.2.

O teste $b = 1$ no final do código, logo após o bloco que calcula o MDC por Euclides, é exatamente o teste que o oráculo fazia na versão anterior. A diferença entre os dois códigos é exatamente o trecho inserido em destaque.

8.2 Amigos quadráticos

Neste problema poderemos explorar a busca por uma solução usando vários níveis de detalhamento, começando pelo pseudo-código de mais alto nível, passando por um outro pseudo-código intermediário e finalmente, através de mais um passo de refinamento, obtendo o código final.

Problema: Dois números naturais n e m são ditos *amigos quadráticos* quando n é igual a soma dos dígitos de m^2 e ao mesmo tempo m é igual a soma dos dígitos de n^2. Imprimir todos os pares n e m que são amigos quadráticos no intervalo $1 \leq 10000 \leq n$ e $1 \leq 10000 \leq m$.

Por exemplo, os números 13 e 16 são amigos quadráticos, pois $13^2 = 169$ e 1+6+9=16. Por outro lado, $16^2 = 256$ e 2+5+6=13.

Novamente usaremos a técnica de se resolver o problema por partes. Vamos iniciar gerando os pares de números dentro do intervalo desejado, conforme fizemos na seção anterior. Para cada par, alguns oráculos nos auxiliarão na busca pela solução. O pseudo-código para isto está na figura 8.3.

Agora podemos nos concentrar exclusivamente no problema de se determinar se um par de números m e n são amigos quadráticos.

Para isto temos que resolver mais um subproblema, pois a definição de amigos

```
program primosentresi;
var i, j, a, b, resto: integer;
begin
        i:= 2;
        while i <= 100 do
        begin
             j:= i;
             while  j <= 100 do
             begin
                  a:= i; b:= j;              (* inicio do bloco euclides  *)
                  resto:= a mod b;           (*           *               *)
                  while resto <> 0 do        (*           *               *)
                  begin                      (*           *               *)
                       a:= b;                (*           *               *)
                       b:= resto;            (*           *               *)
                       resto:= a mod b;      (*           *               *)
                  end;                       (* termino do bloco euclides *)

                 if b = 1 then writeln (i,j);       (* b=1 era o oraculo *)
                 j:= j + 1;
             end;
             i:= i + 1;
        end;
end.
```

Figura 8.2: Gerando todos os primos entre si.

quadráticos exige que a soma dos dígitos de um número ao quadrado tem que ser igual a soma dos dígitos do quadrado do outro número.

Isto envolve decompor o quadrado de um número qualquer na soma dos dígitos que o compõe. Isto pode ser feito como na figura 8.4.

Finalmente temos um terceiro subproblema: como podemos separar os dígitos de um número inteiro qualquer dado, para depois somá-los? A técnica usada explora a operação aritmética de se pegar o resto da divisão inteira por 10. Vejamos um exemplo para o número 169.

```
169 div 10
  9     16 div 10
        6       1 div 10
                1      0
```

No exemplo acima o número 169 foi sucessivamente dividido por 10 até virar zero. Os restos das divisões por 10 nos dão os dígitos do 169, a saber, 9, 6 e 1, obtidos nesta ordem. Isto está implementado conforme ilustrado na figura 8.5.

A variável *digito* contém, a cada iteração, um dígito que faz parte do número dado. Para se obter a soma, basta usar a técnica do acumulador, conforme a figura 8.6.

Por último, não resta mais nada a não ser juntar os diversos pedaços de código em um único programa que executa a tarefa com sucesso. Isto é mostrado na figura 8.7. O leitor deve estudar atentamente a maneira como os diversos códigos foram integrados.

```
begin
      n:= 1;
      while n <= 10000 do
      begin
            m:= 1;
            while m<= 10000 do
            begin
                  if {m e n sao amigos quadraticos} then
                        writeln(n,' e ', m, ' sao amigos quadraticos.');
                  m:= m + 1;
            end;
            n:= n + 1;
      end;
end.
```

Figura 8.3: Pseudo-código para o problema dos amigos quadráticos.

```
begin {dados n e m}
      soma_n:= {oraculo que soma os digitos de n};
      soma_m:= {oraculo que soma os digitos de m};
      if soma_n = soma_m then
            {sao amigos quadraticos}
end.
```

Figura 8.4: Pseudo-código para decidir sobre amigos quadráticos.

É possível contudo fazer menos cálculos e economizar um laço, reduzindo de um milhão para mil operações se, para cada n, testarmos se a soma dos seus dígitos ao quadrado pode ser transformado novamente em n. Veja o algoritmo da figura 8.8 e compare com a versão anterior.

8.3 Palíndromos

Este problema, e o da seção seguinte, nos permitirá entender melhor a técnica usada na seção anterior, a de se decompor um número qualquer na sequência de dígitos que o compõe para depois poder fazer algum cálculo com eles. A ideia é que os dados de cálculo do programa são gerados ao invés de lidos do teclado. Esta é a diferença

```
begin {dado n inteiro}
      while n <> 0 do
      begin
            digito:= n mod 10;
            n:= n div 10;
      end;
end.
```

Figura 8.5: Separando os dígitos de um dado número n.

```
begin {dado N inteiro}
      soma:= 0;
      while N <> 0 do
      begin
            digito:= N mod 10;
            soma:= soma + digito;
            N:= N div 10;
      end;
end.
```

Figura 8.6: Somando os dígitos de um dado número n.

básica.

Problema: Imprimir todos os números palíndromos entre 1 e 1000.

Números palíndromos são aqueles que são lidos da direita para a esquerda da mesma maneira que da esquerda para a direita. Por exemplo o número 12321 é palíndromo, enquanto que 123 não é.

O algoritmo apresentado na figura 8.9 mostra a solução, que exige um aninhamento duplo de comandos de repetição e um teste dentro de um deles. Ele testa todos os números e imprime quando são palíndromos.

Para testar se um número é palíndromo usamos a seguinte técnica: separamos os dígitos do número em questão, em seguida reconstruímos o número ao contrário, usando a técnica do acumulador, e finalmente comparamos com o número original.

Lembrando que um número da forma $d_0 d_1 d_2 \ldots d_n$ é escrito da seguinte forma:

$$d_o \times 10^0 + d_1 \times 10^1 + d_2 \times 10^2 + \ldots + d_n \times 10^n$$

Para gerar o número ao contrário basta fazer multiplicações sucessivas por 10 invertendo-se os dígitos, assim:

$$d_o \times 10^n + d_1 \times 10^{n-1} + d_2 \times 10^{n-2} + \ldots + d_n \times 10^0$$

Testar palíndromos pode ser muito caro dependendo da aplicação. Em todo caso, apresentamos um problema no mínimo divertido, que é o de gerar os palíndromos, assim evitando que todos os números sejam testados. De qualquer forma, este problema serve para o estudante perceber melhor o uso de contadores em diversos níveis de aninhamentos de comandos de repetição.

Problema: Gerar todos os números palíndromos entre 1 e 1000.

O algoritmo apresentado na figura 8.10 contém a solução para este problema. Ele tem por base o formato dos palíndromos, gerando todos os de um dígito, depois todos os de dois dígitos e finalmente todos os de 3 dígitos. Um exercício interessante seria generalizar este código, mas deixamos isto como exercício.

```
program amigosquadraticos;
var n, m, n2, soma_dig_n2, digito, m2, soma_dig_m2: integer;
begin
      n:= 1;
      while n <= 10000 do
      begin
            m:= 1;
            while m<= 10000 do
            begin
                  (* decompoe n ao quadrado *)
                  n2:= n*n;
                  soma_dig_n2:= 0;
                  while n2 <> 0 do
                  begin
                        digito:= n2 mod 10;
                        soma_dig_n2:= soma_dig_n2 + digito;
                        n2:= n2 div 10;
                  end;

                  (* decompoe m ao quadrado *)
                  m2:= m*m;
                  soma_dig_m2:= 0;
                  while m2 <> 0 do
                  begin
                        digito:= m2 mod 10;
                        soma_dig_m2:= soma_dig_m2 + digito;
                        m2:= m2 div 10;
                  end;

                  if (soma_dig_n2 = m) and (soma_dig_m2 = n) then
                        writeln(n,' e ', m, ' sao amigos quadraticos.');
                  m:= m + 1;
            end;
            n:= n + 1;
      end;
end.
```

Figura 8.7: Verificando se números n e m são amigos quadráticos.

8.4 Inverter um número de três dígitos

O próximo problema vai nos permitir reforçar o uso da técnica do acumulador ao mesmo tempo em que continuamos a mostrar várias soluções para o mesmo problema de maneira a evoluirmos na compreensão da arte de se construir algoritmos.

Problema: Ler um número de 3 dígitos do teclado e imprimir este número invertido. Exemplo, se a entrada for "123" (cento e vinte e três) a saída deve ser "321" (trezentos e vinte e um).

A primeira solução, apresentada na figura 8.11, foi feita pelos alunos em sala de aula em um curso no segundo semestre de 2002. É um bom exemplo de como particularizar a solução leva, quase que necessariamente, a dificuldades imensas na reimplementação do código para problemas similares.

Os estudantes propuseram esta solução por considerarem que o ser humano faz mais ou menos do mesmo modo, isto é, ele separa os dígitos mentalmente lendo o número de trás para frente ao mesmo tempo que os escreve na ordem desejada.

A solução então parte do princípio de, primeiro separar o número dado na entrada em três dígitos para, depois, imprimi-los na ordem inversa. Assim, os operadores de divisão e resto de divisão inteira são utilizados para a separação dos números em unidade, dezena e centena.

O programa funciona, é verdade, mas tem um grave problema. Ele só funciona para números com 3 dígitos. Se fosse pedido para resolver o mesmo problema para um número de 4 dígitos, então a implementação do mesmo método implicaria na total reconstrução do código acima. Isto é, é como se tivéssemos um novo problema para ser resolvido e nunca tivéssemos pensado em nenhuma solução parecida, o que não é verdade.

O código da figura 8.12 ilustra a adaptação desta mesma ideia, desta vez para 4 dígitos.

Além de ter que declarar mais uma variável (milhar), houve alteração de praticamente todas as linhas do código. Aparentemente, separar a unidade e o primeiro dígito (no caso o milhar) é fácil, bastando uma divisão inteira por 10 e um resto de divisão inteira por 10. Mas calcular a dezena e a centena começa a ficar complicado.

É possível imaginar o estrago no código se for exigido que a entrada seja constituída de 10 dígitos ou mais, o que nos permite perceber claramente a dificuldade na adaptação da solução ao mesmo tempo em que se percebe que, para o computador, nem sempre é fácil resolver problemas da mesma maneira como o ser humano faria.

Novamente, é preciso deixar que o computador faça o trabalho, não o programador. Se este último for esperto o suficiente, fará um código baseado num algoritmo mais elaborado. Provavelmente o algoritmo não será mais tão intuitivo, mas se for bem feito, vai nos permitir generalizar o código para entradas de qualquer tamanho.

O que queremos aqui, assim como no problema da seção anterior, é tentar perceber algum raciocínio que possa ser repetido um número controlado de vezes. Deve-se tentar explorar uma mesma sequência de operações, que consiste em separar o último dígito e remover do número original este dígito que foi separado.

Desta forma, a construção da resposta não pode mais ser feita apenas no final do

algoritmo. À medida que o número original vai sendo separado, o seu inverso já deve ir sendo construído, tal como foi feito nas duas seções anteriores. O código final é apresentado na figura 8.13.

8.5 Cálculo do MDC pela definição

Nesta seção apresentaremos a solução do problema do MDC calculado pela definição. O objetivo é motivar o capítulo seguinte, uma vez que sabemos que existe um algoritmo melhor, estudado na seção 6.16.

O MDC entre dois inteiros a e b foi definido matematicamente na seção 6.4.1 e envolve a fatoração de ambas as entradas como um produto de números primos. O algoritmo básico, em pseudo-código é apresentado na figura 8.14.

O princípio do algoritmo é verificar quantas vezes cada número primo divide as entradas e descobrir qual deles é o menor. O MDC é atualizado então para menor potência deste primo. É preciso separar o caso do 2 dos outros, por motivos que já discutimos. Os valores de a e de b são atualizados, para não haver cálculos inúteis com os ímpares múltiplos de primos que já foram previamente processados.

O programa que implementa este algoritmo não cabe em uma página, por isto é apresentado em duas partes nas figuras 8.15 e 8.16.

Neste ponto deixamos claro ao leitor o motivo da apresentação deste problema no final deste capítulo: este código tem um nível muito alto de trechos de código bastante parecidos. Observamos que, em quatro vezes, se calcula quantas vezes um dado primo p divide um número n. Ainda, a atualização do MDC também aparece em dois trechos diferentes mas bastante similares.

O reaproveitamento de código é uma das motivações para o uso de *subprogramas* nas linguagens de alto nível. Mas não é a única, existem outras motivações. No próximo capítulo vamos introduzir as importantes noções de *procedure* e *function* em *Pascal*, e poderemos reescrever o código acima com muito mais elegância.

```
program amigosquad;
var n, m, n2, m2, soma_dig_n2, soma_dig_m2, unidade: integer;

begin
      n:= 1;
      while n <= 10000 do
      begin
            (* decompoe n ao quadrado *)
            n2:= n*n;
            soma_dig_n2:= 0;
            while n2 <> 0 do
            begin
                  unidade:= n2 mod 10;
                  soma_dig_n2:= soma_dig_n2 + unidade;
                        n2:= n2 div 10;
            end;

            m:= soma_dig_n2;
            if soma_dig_n2 <= 1000 then (* se estiver na faixa permitida *)
            begin
                  (* decompoe soma_dig_n2 *)
                  m2:= m*m;
                  soma_dig_m2:= 0;
                  while m2 <> 0 do
                  begin
                        unidade:= m2 mod 10;
                        soma_dig_m2:= soma_dig_m2 + unidade;
                        m2:= m2 div 10;
                  end;

                  if soma_dig_m2 = n then
                        writeln(n,' e ', m, ' sao amigos quadraticos.');
            end;
            n:= n + 1;
      end;
end.
```

Figura 8.8: Tornando amigos quadráticos mais eficiente.

```
program todospalindromos;
const max=1000;
var i, invertido, n: integer;
begin
      i:= 1;
      while i <= max do (* laco que controla os numeros entre 1 e max *)
      begin
            invertido:= 0; (* inicializa aculumador *)
            n:= i;

            while n <> 0 do
            begin
                  invertido:= invertido*10 + n mod 10;
                  n:= n div 10;
            end;

            (* imprime se for palindromo, senao nao faz nada *)
            if invertido = i then
                  writeln (i);

            i:= i + 1;
      end;
end.
```

Figura 8.9: Imprimindo todos os palíndromos de 1 a 1000.

```
program gerandopalindromos;
var i, j, pal: integer;
begin
      i:= 1;                              (* gerando todos de um digito *)
      while i <= 9 do
      begin
            writeln (i);
            i:= i + 1;
      end;

      pal:= 11;                           (* gerando todos de 2 digitos *)
      i:= 2;
      while i <= 10 do
      begin
            writeln (pal);
            pal:= i * 11;
            i:= i + 1;
      end;


      i:= 1;                            (* gerando todos os de 3 digitos *)
      while i <= 9 do
      begin
            j:= 0;
            while j <= 9 do
            begin
                  pal:= i*100 + j*10 + i;
                  writeln (pal);
                  j:= j + 1;
            end;
            i:= i + 1;
      end;
end.
```

Figura 8.10: Gerando todos os palíndromos de 1 a 1000.

```
program inverte3_v0;
var numero, unidade, dezena, centena, inverso: integer;
begin
      write('entre com o numero de tres digitos: ');
      readln(numero);
      centena:= numero div 100;
      dezena:= (numero mod 100) div 10;
      unidade:= numero mod 10;
      inverso := unidade*100 + dezena*10 + centena;
      writeln(inverso);
end.
```

Figura 8.11: Primeira solução para inverter um número de 3 dígitos.

```
program inverte3_v0;
var numero, unidade, dezena, centena, milhar, inverso: integer;
begin
      write('entre com o numero de quatro digitos: ');
      readln(numero);
      milhar:= numero div 1000;
      centena:= (numero mod 1000) div 100;
      dezena:= (numero mod 100) div 10;
      unidade:= numero mod 10;
      inverso := unidade*1000 + dezena*100 + centena*10 + milhar;
      writeln(inverso);
end.
```

Figura 8.12: Mesmo algoritmo, agora para 4 dígitos.

```
program inverte3_v1;
var i, numero, unidade, inverso, resto: integer;
begin
      write('entre com o numero de tres digitos: ');
      readln(numero);
      inverso := 0;

      i:= 1;
      while (i <= 3) do
      begin
          unidade := numero mod 10;
          resto := numero div 10;
          inverso := inverso*10 + unidade;
          numero := resto;
          i:= i + 1;
      end;

      writeln(inverso);
end.
```

Figura 8.13: Solução com uso de acumuladores.

```
begin
      read (a,b);
      mdc:= 1;

      (* descobre quantas vezes o 2 divide as duas entradas *)
      cont_a:= {numero de vezes que o 2 divide a};
      cont_b:= {numero de vezes que o 2 divide b};
      menor_cont:= {menor entre cont_a e cont_b};
      mdc:= mdc * {2 elevado a potencia menor_cont};
      a:= a div mdc;
      b:= b div mdc;

      (* repete o processo para todos os impares *)
      primo:= 3;
      while (a <> 1) and (b <> 1) do
      begin
            cont_a:= {numero de vezes que primo divide a}
            cont_b:= {numero de vezes que primo divide b}
            menor_cont:= {menor entre cont_a e cont_b};
            mdc:= mdc * {primo elevado a potencia menor_cont};
            a:= a div mdc;
            b:= b div mdc;
            primo:= primo + 2;          (* passa para o proximo impar *)
      end;
      writeln (mdc);
end.
```

Figura 8.14: Pseudo-código para o calculo do MDC pela definição.

```
program mdc_por_definicao;
var i, a, b, mdc, cont_a, cont_b, menor_cont, primo: integer;
begin
      (* inicializacao das variaveis principais *)
      read (a,b);
      mdc:= 1;

      (* descobre quantas vezes o 2 divide as duas entradas *)
      cont_a:= 0;
      while a mod 2 = 0 do
      begin
            cont_a:= cont_a + 1;
            a:= a div 2;
      end;

      cont_b:= 0;
      while b mod 2 = 0 do
      begin
            cont_b:= cont_b + 1;
            b:= b div 2;
      end;

      (* descobre qual dos contadores eh o menor *)
      if cont_a <= cont_b then
            menor_cont:= cont_a
      else
            menor_cont:= cont_b;

      (* atualiza o mdc para o 2 *)
      i:= 1;
      while i <= menor_cont do
      begin
            mdc:= mdc * 2;
            i:= i + 1;
      end;
```

Figura 8.15: Calcula MDC entre a e b pela definição (caso primo=2).

```
      (* repete o processo para todos os impares *)
      primo:= 3;
      while (a <> 1) and (b <> 1) do
      begin
            cont_a:= 0;
            while a mod primo = 0 do
            begin
                  cont_a:= cont_a + 1;
                  a:= a div primo;
            end;

            cont_b:= 0;
            while b mod primo = 0 do
            begin
                  cont_b:= cont_b + 1;
                  b:= b div primo;
            end;

            (* descobre qual dos contadores eh o menor *)
            if cont_a <= cont_b then
                  menor_cont:= cont_a
            else
                  menor_cont:= cont_b;

            (* atualiza o mdc para o primo impar da vez *)
            i:= 1;
            while i <= menor_cont do
            begin
                  mdc:= mdc * primo;
                  i:= i + 1;
            end;

            (* passa para o proximo impar *)
            primo:= primo + 2;
      end;

      (* imprime o resultado final *)
      writeln (mdc);
end.
```

Figura 8.16: Calcula MDC entre a e b pela definição (caso primo é ímpar).

8.6 Exercícios

1. Fazer um programa em *Pascal* que leia do teclado dois números inteiros positivos e que imprima na saída um único número inteiro que é a soma dos dois primeiros. Entretanto, seu programa não pode utilizar o operador de soma (+) da linguagem *Pascal* para somar os dois inteiros lidos em uma única operação. Outrossim, o programa deve implementar a soma dos números dígito a dígito, iniciando pelo menos significativo até o mais significativo, considerando o "vai um", conforme costumamos fazer manualmente desde o ensino fundamental.

```
Exemplo 1                    Exemplo 2
 11   ("vai um")              1111   ("vai um")
40912 (primeiro número)        52986 (primeiro número)
 1093 (segundo número)       1058021 (segundo número)
-----                        -------
42005 (soma)                 1111007 (soma)
```

2. Um agricultor possui 1 (uma) espiga de milho. Cada espiga tem 150 grãos, e cada grão pesa 1g (um grama). Escreva um programa em *Pascal* para determinar quantos anos serão necessários para que o agricultor colha mais de cem toneladas de milho (1T = 1000Kg, 1Kg = 1000g), sendo que:

   - A cada ano ele planta todos os grãos da colheita anterior
   - Há apenas uma colheita por ano
   - 10% (dez por cento) dos grãos não germina (morre sem produzir)
   - Cada grão que germina produz duas espigas de milho

   Assuma que a quantidade de terra disponível é sempre suficiente para o plantio.

3. Modifique a questão anterior acrescentando na simulação os seguintes fatos:

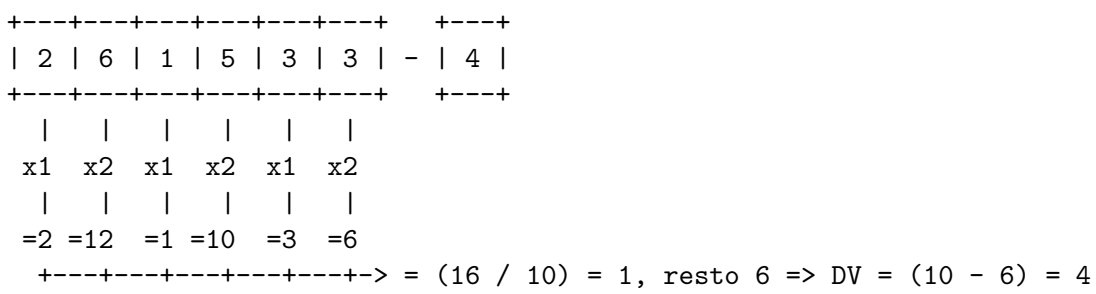
   - Há 8 (oito) CASAIS de pombas (16 pombas) que moram na propriedade do agricultor.
   - Cada pomba come 30 grãos por dia, durante os 30 dias do ano em que as espigas estão formadas antes da colheita;
   - A cada ano, cada casal gera 2 novos casais (4 pombas), que se alimentarão e reproduzirão no ano seguinte;
   - Uma pomba vive tres anos;

   Ao final do programa, imprima também o número de pombas que vivem na propriedade quando o agricultor colher mais de 100T de milho

4. Considere um número inteiro com 9 dígitos. Suponha que o último dígito seja o "dígito verificador" do número formado pelos 8 primeiros. Faça um programa em *Pascal* que leia uma massa de dados terminada por 0 (zero) e que imprima os números que não são bem formados, isto é, aqueles que não satisfazem o dígito verificador. Implemente o seguinte algoritmo para gerar o dígito verificador:

   Conforme o esquema abaixo, cada dígito do número, começando da direita para a esquerda (menos significativo para o mais significativo) é multiplicado, na ordem, por 2, depois 1, depois 2, depois 1 e assim sucessivamente.

   Número exemplo: 261533-4



   Ao invés de ser feita a somatória das multiplicações, será feita a somatória dos dígitos das multiplicações (se uma multiplicação der 12, por exemplo, será somado $1 + 2 = 3$).

   A somatória será dividida por 10 e se o resto (módulo 10) for diferente de zero, o dígito será 10 menos este valor.

5. Escreva um programa *Pascal* que leia dois valores inteiros positivos A e B. Se A for igual a B, devem ser lidos novos valores até que sejam informados valores distintos. Se A for menor que B, o programa deve calcular e escrever a soma dos números ímpares existentes entre A(inclusive) e B(inclusive). Se A for maior que B, o programa deve calcular e escrever a média aritmética dos múltiplos de 3 existentes entre A(inclusive) e B(inclusive).

6. Faça um programa em *Pascal* que dado uma sequência de números inteiros terminada por zero (0), determinar quantos segmentos de números iguais consecutivos compõem essa sequência.

   Ex.: A sequência 2,2,3,3,5,1,1,1 é formada por 4 segmentos de números iguais.

7. Faça um programa em *Pascal* que imprima a seguinte sequência de números: 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, ...

8. Faça um programa em *Pascal* que receba como entrada um dado inteiro N e o imprima como um produto de primos. Exemplos: $45 = 3 \times 3 \times 5$. $56 = 2 \times 2 \times 2 \times 7$.

9. Faça um programa em *Pascal* que, dado um número inteiro N, escreva o maior divisor de N que é uma potência de um dos números primos fatorados. Ex:

   $N = 45 = 3^2.5^1$ escreve $9 = 3^2$

   $N = 145 = 5^2.7^1$ escreve $25 = 5^2$

   $N = 5616 = 2^4.3^3.13$ escreve $27 = 3^3$

Capítulo 9

Funções e procedimentos

Até agora vimos as noções básicas de algoritmos, fundamentalmente como funciona o computador de verdade (modelo Von Neumann) e como isto pode ter uma representação em um nível mais alto, guardadas as limitações da máquina.

A maneira de se resolver problemas do ponto de vista algoritmico envolve a elaboração de construções com apenas quatro tipos de estruturas de controle de fluxo: comandos de entrada e saída, comandos de atribuição, comandos de repetição e comandos de desvio condicional, além do uso de expressões lógicas e aritméticas. Para a efetiva programação ainda é necessário dominar a arte de escrever os algoritmos em função das limitações dos compiladores.

O problema é que à medida em que os problemas exigem códigos com muitas linhas, os programas gerados vão se tornando cada vez mais complexos tanto para serem desenvolvidos quanto para serem mantidos em funcionamento. O exemplo do final do capítulo anterior dá uma ideia dessa dificuldade.

Por isto, os programas devem ser construídos de maneira a facilitar o trabalho dos programadores e analistas. É preciso que os códigos sejam elaborados com partes bem definidas, em módulos que contenham pedaços coerentes do problema geral que se está tentando modelar. As noções de funções e procedimentos são o caminho para se construir códigos elegantes, robustos e de fácil manutenção.

9.1 Motivação

Os procedimentos e funções são nada mais do que subprogramas, isto é, pedaços de programas dentro de programas. Mas, bem explorados, permitem a construção de códigos altamente elaborados. Existem três motivações para se usar subprogramas:

- modularidade;
- reaproveitamento de código;
- legibilidade do código.

Vamos detalhar cada tópico nos próximos itens.

9.1.1 Modularidade

A noção de modularidade é relacionada com a capacidade de se escrever programas em pedaços de código que executam operações bem definidas. Cada módulo possui variáveis e estruturas próprias, independentes do restante do programa. A ideia é que modificações em trechos de código (necessárias para manutenção e continuidade de desenvolvimento) não causem reflexos no comportamento do resto do programa.

É fácil conceber um programa dividido em três partes: entrada de dados, cálculos e impressão dos resultados. Uma entrada de dados textual poderia ser modificada para outra em modo gráfico sem que o pedaço que faz os cálculos perceba a modificação. Idem para a saída de dados. Mesmo nos cálculos, pode-se mudar toda uma estrutura do programa e garantir que a entrada e saída vão continuar a se comportar como antes. Isto é tarefa árdua sem o uso de funções e procedimentos.

É importante notar que a linguagem *Pascal* não fornece todos os meios para se implementar um programa totalmente modular, mas é o suficiente para os estudantes em primeiro curso de computação perceberem a importância do conceito. Na verdade, a evolução destas ideias deu origem ao conceito de Programação Orientada a Objetos, hoje na moda. Mas isto é tema para outras disciplinas.

9.1.2 Reaproveitamento de código

Vez por outra nos deparamos com situações onde temos que escrever códigos muito, mas muito, parecidos em trechos diferentes do programa. As vezes a diferença de um para outro é questão de uma ou outra variável que muda. Ocorre frequentemente que o trecho é exatamente o mesmo.

Então faz sentido que possamos estruturar o código repetido de maneira a constituir um subprograma e, no programa propriamente dito, fazer o código do subprograma ser executado para diferentes valores de variáveis. Isto provoca uma grande economia de código escrito, ao mesmo tempo em que facilita a manutenção do programa.

9.1.3 Legibilidade

Os dois aspectos acima, somados com o bom uso de nomes apropriados para os identificadores, endentação e uso racional de comentários no código, implicam necessariamente em um código legível, isto é, compreensível para quem o lê e até mesmo para quem o escreveu.[1]

De fato, é comum alguém fazer um programa, as vezes simples, e depois de alguns meses ler o código e não entender o que lá está escrito. Um código legível permite uma rápida compreensão e viabiliza sua manutenção, correção e expansão, seja pelo próprio programador ou por outras pessoas.

[1] Recomendamos a leitura do mini-guia da linguagem *Pascal*, diponível no site oficial da disciplina CI055.

9.1.4 Comentário adicional

Nesta seção, vamos tentar convencer o aprendiz a usar bem e a dar valor a estas noções, estudando exemplos simples, mas didáticos.

9.2 Noções fundamentais

Existem três conceitos que devem ser dominados pelo estudante:

1. quando usar *função* e quando usar *procedimento*?
2. quando usar *variáveis locais* ou *variáveis globais*?
3. quando usar passagem de parâmetros *por valor* ou *por referência*?

Nas próximas seções vamos detalhar cada um destes itens.

9.2.1 Exemplo básico

Vamos tomar como base um programa bem simples, estudado nas primeiras aulas desta disciplina. Trata-se do problema de se ler uma sequência de valores inteiros terminada por zero e que imprima somente aqueles que são pares.

Quando resolvemos este problema, o código foi escrito como na figura 9.1.

```
program imprime_pares;
var a: integer;
begin
      read (a);
      while a <> 0 do
      begin
            if a mod 2 = 0 then
                  writeln (a);
            read (a);
      end;
end.
```

Figura 9.1: Programa que imprime os números da entrada que são pares.

9.2.2 O programa principal

É o código do programa propriamente dito. Tem início no *begin* e término no *end*.

No exemplo anterior, são todos os comandos que aparecem no programa, desde o de leitura da variável *a* até o *end*; do comando *while*. O resto que lá aparece é o cabeçalho do programa e a declaração de variáveis globais.

9.2.3 Variáveis globais

São todas as variáveis declaradas logo após o cabeçalho do programa, antes do *begin* do programa principal.

Como sabemos, variáveis são abstrações de endereços de memória. As variáveis globais são endereços visíveis em todo o programa, mesmo nos subprogramas, como veremos logo mais.

No exemplo acima, existe uma única variável global, ela tem o identificador a e é do tipo *integer*.

9.2.4 Funções

No miolo do programa exemplo, existe um trecho onde se lê: $a \bmod 2 = 0$. Hoje sabemos que isto é uma expressão booleana que calcula o resto da divisão inteira de a por 2, se o resultado for zero então a é par, senão é impar. Mas, esta expressão poderia ser bem mais complexa, exigindo muitas linhas de código, por exemplo, se quiséssemos imprimir os números que fossem primos ao invés dos pares.

O importante é que a expressão booleana resulta em um valor do tipo booleano. O mesmo efeito pode ser conseguido com uma função que resulta em um valor do tipo booleano.

Isto pode ser feito definindo-se um subprograma que é chamado pelo programa principal e que recebe os dados necessários para os cálculos (os parâmetros). Os cálculos devem computar corretamente se o número enviado pelo programa principal é par. Se for, de alguma maneira deve retornar um valor *true* para quem chamou. Se não for, deve retornar *false*.

O que acaba de ser escrito estabelece uma espécie de "contrato" entre o programa principal e o subprograma. Em termos de linguagem de programação, este contrato é denominado de *protótipo* ou *assinatura* da função. Ele é constituído por três coisas:

- O nome da função;
- A lista de parâmetros, que são identificadores tipados (no caso da linguagem *Pascal*);
- O tipo do valor de retorno contendo o cálculo feito na função.

Para o problema em questão, vamos assumir que poderíamos estabelecer o seguinte protótipo para a função que calcula se um dado número é par retornando *true* em caso positivo e *false* em caso contrário. Vejamos como fica:

```
function a_eh_par: boolean;
```

A palavra reservada *function* é para avisar o programa que se trata de uma função. O identificador a_eh_par procura representar o fato de que a variável a pode ser par.

O código da função não importa muito no momento, mas apenas com o protótipo é possível reescrever o programa exemplo como apresentado na figura 9.2.

```
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function a_eh_par: boolean;
begin
     (* codigo da funcao, ainda nao escrito por razoes didaticas *)
end;

begin (* programa principal *)
     read (a);
     while a <> 0 do
     begin
          if a_eh_par then
               writeln (a);
          read (a);
     end;
end.
```

Figura 9.2: Programa que imprime os números da entrada que são pares.

Desde que o código da função seja corretamente implementado, o programa principal continua funcionando! Além disto é um cógido mais legível, já que é mais simples para alguém ler o programa principal e perceber o significado do identificador denominado a_eh_par ao invés do obscuro "$a \bmod 2 = 0$".

Obviamente é necessário em algum momento se escrever o código da função. Vamos mostrar quatro maneiras diferentes de se devolver o valor correto para o programa principal.

```
(* primeira versao da funcao *)
if a mod 2 = 0 then
     a_eh_par:= true
else
     a_eh_par:= false;
```

```
(* segunda versao da funcao *)
if a mod 2 <> 0 then
     a_eh_par:= false
else
     a_eh_par:= true;
```

```
(* terceira versao da funcao *)
if a mod 2 <> 1 then
     a_eh_par:= true
else
     a_eh_par:= false;
```

```
(* quarta versao da funcao *)
if a mod 2 = 1 then
     a_eh_par:= false
else
     a_eh_par:= true;
```

```
(* quinta versao da funcao *)
a_eh_par:= false;
if a mod 2 = 0 then
     a_eh_par:= true
```

```
(* sexta versao da funcao *)
a_eh_par:= true;
if a mod 2 = 1 then
     a_eh_par:= false
```

```
(* setima versao da funcao *)
a_eh_par:= true;
if a mod 2 <> 1 then
    a_eh_par:= true
```

```
(* oitava versao da funcao *)
a_eh_par:= true;
if a mod 2 <> 0 then
    a_eh_par:= false
```

Em tempo, um detalhe da linguagem *Pascal*. Segundo a definição da linguagem, estabelecida por seu autor Niklaus Wirth ainda nos anos 1970, a maneira como o valor do retorno é feita para o programa principal exige que o nome da função apareça pelo menos uma vez no código da função do lado *esquerdo* de um comando de atribuição. Isto funciona diferente em outras linguagens de programação, para isto, se o estudante resolver mudar de linguagem, deve observar o respectivo guia de referência.

Ainda um comentário adicional sobre a linguagem *Pascal*, a função *é executada até o final, isto é, até encontrar o comando* end*; que a termina*. Isto pode ser diferente em outras linguagens.

9.2.5 Parâmetros por valor

Conforme dissemos, atendemos a questão da modularidade ao usarmos a função a_eh_par. Mas não atendemos a outra: a questão do reaproveitamento de código.

De fato, suponhamos que o enunciado tivesse estabelecido que seriam dados como entrada *pares de números a e b* e para imprimir apenas os que fossem pares. Na versão atual do programa, teríamos que escrever uma outra função de nome provável b_eh_par que teria o código absolutamente idêntico ao da função a_eh_par exceto pela troca da variável *a* pela variável *b*. Isto é inadmissível em programação de alto nível!

Logo, deve existir uma maneira melhor para se programar a função. Felizmente existe: basta passar um parâmetro, isto é, basta informar à função que os cálculos serão feitos para um dado número inteiro, não importando o nome da variável que contém o valor sobre o qual será feito o cálculo da paridade. Isto é conseguido mudando-se o protótipo da função para o seguinte:

```
function eh_par (n: integer): boolean;
```

Em outras palavras, a função vai receber um número inteiro, no caso denominado *n* (mas poderia ser *a*, *b* ou qualquer outro identificador, o importante é que seja do tipo *integer*. O tipo do retorno é o mesmo, isto é, *boolean*.

Com isto, conseguimos escrever o programa (já com a função recebendo parâmetros por valor), da maneira como está apresentado na figura 9.3.

Esta maneira de passar parâmetros caracteriza uma passagem *por valor*, também conhecida como passagem de parâmetros *por cópia*. O que ocorre é que o identificador *n* da função recebe uma cópia do valor da variável *a* do programa principal. As alterações em *n* são feitas nesta cópia, mantendo-se intactos os dados da variável *a* no programa principal. Isto ocorre pois esta cópia é feita em área separada de memória, denominada *pilha*.

```
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function eh_par (n: integer): boolean;
begin
      if n mod 2 = 0 then
            eh_par:= true
      else
            eh_par:= false;
end;

begin (* programa principal *)
      read (a);
      while a <> 0 do
      begin
            if eh_par (a) then
                  writeln (a);
            read (a);
      end;
end.
```

Figura 9.3: Programa que imprime os números da entrada que são pares.

9.2.6 Parâmetros por referência

Para passar parâmetros por referência, o protótipo da função difere ligeiramente daquele exibido acima. A chamada é assim:

```
function eh_par (var n: integer): boolean;
```

A diferença sintática é caracterizada pela palavra *var* antes do nome do identificador do parâmetro.[2] Pode parecer sutil, mas com uma semântica totalmente diferente daquela da passagem de parâmetros por valor.

O que antes era uma cópia, agora é uma referência ao endereço da variável do programa principal. Isto é, no momento da ativação da função, o identificador n da função é associado com o mesmo endereço da variável a no programa principal. Consequentemente, qualquer alteração associada a n na função provocará alteração do valor da variável a no programa principal.

Vejamos na figura 9.4 como fica a modificação no programa em estudo do ponto de vista meramente sintático. Deixamos o ponto de vista semântico para ser explicado no próximo exemplo.

[2]Com todo o respeito ao Niklaus Wirth, usar a palavra *var* para este fim não foi uma boa escolha, pois para um principiante é fácil confundir uma passagem de parâmetros por referência com uma declaração de uma variável, o que não é definitivamente o caso aqui.

```
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function eh_par (var n: integer): boolean;
begin
      if n mod 2 = 0 then
            eh_par:= true
      else
            eh_par:= false;
end;

begin (* programa principal *)
      read (a);
      while a <> 0 do
      begin
            if eh_par (a) then
                  writeln (a);
            read (a);
      end;
end.
```

Figura 9.4: Versão com parâmetros por referência.

9.2.7 Procedimentos

Um procedimento difere de uma função basicamente pois não tem um valor de retorno associado. Isto faz com que ele se comporte como um comando extra da linguagem ao passo que a função tem um comportamento mais parecido com o de uma expressão aritmética ou booleana. Um protótipo para um procedimento é definido então com base em apenas duas informações:

1. o identificador do procedimento (nome);
2. a lista de parâmetros (que podem ser por valor ou referência).

Para explicar corretamente a noção de procedimentos, e também de variáveis locais, é importante mudar de exemplo. Vamos considerar o problema de se ler dois números reais x e y do teclado, fazer a troca dos conteúdos e depois imprimir.

Vamos considerar o seguinte protótipo para um procedimento que recebe os dois valores x e y e tem a finalidade de realizar a troca dos conteúdos destas variáveis:

```
procedure troca (var a, b: real);
```

Observe que, assim como no caso da função, não importa muito o nome dos identificadores dos parâmetros, apenas importa que eles definam conteúdos do tipo REAL.

Podemos então tentar escrever um programa que leia dois valores e os imprima trocados, conforme ilustrado na figura 9.5[3]:

[3]Um exercício interessante é passar os parâmetros por valor e compreender o efeito disso.

```
program imprimetrocado;
var x,y,temp: real; (* variaveis globais *)

(* procedimento que troca os conteudos da variaveis *)
procedure troca (var a, b: real);
begin
      temp:= a;
      a:= b;
      b:= temp;
end;

begin (* programa principal *)
      read (x,y);
      troca (x,y);
      writeln (x,y);
end.
```

Figura 9.5: Versão com parâmetros por referência.

Este programa usa uma variável global (*temp*) para auxiliar a troca, o que não faz muito sentido. Os subprogramas devem funcionar independentemente de variáveis globais.

9.2.8 Variáveis locais

Variáveis locais são declaradas nos subprogramas e têm escopo local, isto é, elas só são conhecidas durante a execução do subprograma. Consequentemente não interferem no programa principal. O programa modificado da figura 9.6 faz uso da variável local *temp* e torna o código mais robusto.

```
program imprimetrocado;
var x,y: real; (* variaveis globais *)

(* procedimento que troca os conteudos da variaveis *)
procedure troca (var a, b: real);
var temp: real; (* variavel local, temporaria para uso exclusivo neste procedimento *)
begin
      temp:= a;
      a:= b;
      b:= temp;
end;

begin (* programa principal *)
      read (x,y);
      troca (x,y);
      writeln (x,y);
end.
```

Figura 9.6: Versão com uma variável local.

9.3 Alguns exemplos

Nesta seção vamos implementar dois problemas simples para exercitar.

9.3.1 Calculando dígito verificador

Vamos fazer um programa que recebe um número de N dígitos, sendo o último deles o "dígito verificador" do número formado pelos $N-1$ primeiros. Devemos calcular se o dígito verificador fornecido pelo usuário está correto segundo o esquema de cálculo seguinte: cada dígito do número, começando da direita para a esquerda (menos significativo para o mais significativo) é multiplicado, na ordem, por 1, depois 2, depois 1, depois 2 e assim sucessivamente. O número de entrada do exemplo é 261533-4.

```
+---+---+---+---+---+---+   +---+
| 2 | 6 | 1 | 5 | 3 | 3 | - | 4 |
+---+---+---+---+---+---+   +---+
  |   |   |   |   |   |
 x2  x1  x2  x1  x2  x1
  |   |   |   |   |   |
 =4  =6  =2  =5  =6  =3
  +---+---+---+---+---+-> = 26
```

Como 26 tem dois dígitos, vamos repetir o processo acima até gerarmos um número de um único dígito. Assim:

```
+---+---+
| 2 | 6 |
+---+---+
  |   |
 x2  x1
  |   |
 =4  =6
  +---+ = 10
```

Como 10 ainda tem dois dígitos, o algoritmo roda ainda mais uma vez:

```
+---+---+
| 1 | 0 |
+---+---+
  |   |
 x2  x1
  |   |
 =2  =0
  +---+ = 2
```

Assim, o dígito verificador calculado (2) difere daquele fornecido (4) e o programa deveria acusar o erro. O programa da figura 9.7 ilustra uma possível solução.

```
program digitoverificador;
var numero, n: longint;
    dv, dv_correto: integer;

procedure le (var n: longint);
begin
     {$i-}
     repeat
          read (n);
     until ioresult = 0;
     {$i+}
end;

procedure separa_numero_e_dv (n: longint; var num: longint; var dv: integer);
begin
     num:= n div 10;
     dv:=  n mod 10;
end;

function calcula_dv_correto (n: longint): integer;
var soma, mult, ultimo: integer;
begin
     repeat
          soma:= 0;
          mult:= 1;
          while n <> 0 do
          begin
               ultimo:= n mod 10;
               n:= n div 10;
               soma:= soma + mult * ultimo;
               if mult = 1 then
                    mult:= 2
               else
                    mult:= 1;
          end;
          n:= soma;
     until (n > 0) and (n <= 9);
     calcula_dv_correto:= soma;
end;

begin (* programa principal *)
     le (numero);
     separa_numero_e_dv (numero, n, dv);
     dv_correto:= calcula_dv_correto (n);
     if dv_correto <> dv then
          writeln ('digito verificador invalido.')
end.
```

Figura 9.7: Calculando dígito verificador.

O importante para se observar neste código é a clareza do algoritmo no programa principal. O leitor pode acompanhar este trecho e perceber claramente as diversas etapas em uma linguagem de bastante alto nível: leitura do número, separação deste em duas partes, uma contendo os primeiros dígitos a outra contendo o dv de entrada. Em seguida o cálculo do dígito verificador correto e finalmente a comparação dos dados calculados com o de entrada, gerando a mensagem final.

No programa principal pode-se ignorar completamente como são feitas todas as operações nas funções e procedumentos: não importa como os dados são lidos, nem como os dígitos são separados, e muito menos como é feito o cálculo do dígito verificador correto. No programa principal o importante é o algoritmo em alto nível.

É claro que em algum momento será necessário escrever código para cada uma das funções e procedimentos, mas quando isto for feito o programador estará resolvendo um subproblema de cada vez, o que facilita muito a construção do código para o problema global.

Por exemplo, a leitura poderia ser feita em uma interface gráfica ou textual. Foi escolhida nesta versão uma interface textual, mas que permite testes de consistência dos dados de entrada. Isto, feito separadamente, mantém o código global independente. No caso, o programador usou diretivas do compilador para alterar o comportamento padrão que aborta o programa se o usuário digitar um tipo não esperado. Trata corretamente o erro e quando tudo está certo, termina o procedimento. Importante notar que, para leitura de dados, o parâmetro tem que ser passado por referência, e não por valor, senão o valor seria lido em uma cópia da variável do programa principal.

Considerações similares são feitas para a outra função e o procedimento, ambos possuem código próprio e fazem com que a atenção do programador fique voltada exclusivamente para o subproblema da vez. Desde que o protótipo da função ou procedimento esteja corretamente especificado, não há problema em se alterar o código interno. Notamos também que no caso da função, foram usadas variáveis locais para auxílio no cálculo do dígito verificador. Tudo para o bem da clareza do código principal. Se um dia mudarem a definição de como se calcula o dígito verificador, basta alterar a função que o programa principal continuará a funcionar corretamente.

Na última função é importante notar a passagem de parâmetros por valor. Isto permitiu o laço que controla o loop interno usar o próprio parâmetro como condição de parada, pois ele é dividido por 10 a cada iteração. Se o parâmetro fosse passado por referência isto não poderia ser feito, pois estaríamos estragando o valor da variável que será ainda usada no programa principal.

Ainda uma última observação, no procedimento que separa o número de seu dígito verificador, atentar para a sintaxe, em *Pascal*, que se usa quando se quer passar dois parâmetros do mesmo tipo, um por valor e outro por referência. No caso, eles devem ser separados pelo símbolo do ponto-e-vírgula, o segundo deles contendo a palavra *var* para indicar referência.

9.4 Cálculo do MDC pela definição

Nesta seção vamos revisitar o cálculo do MDC pela definição estudado na seção 8.5. Deixamos propositalmente em aberto a questão de que aquele código continha trechos de código similares que tinham que ser repetidos por causa de um número de entrada variante no programa.

De fato, naquele código exibido na figura 8.15 o cálculo para saber se o número a era par difere do cálculo para saber se o número b é par apenas por conta do valor de a ou b. O mesmo ocorre para o código da figura 8.16 no caso de números ímpares.

Na verdade, os quatro trechos de código estão ali apenas para se saber quantas vezes um dado número n pode ser dividido por um número primo p, seja ele o 2 ou qualquer ímpar primo.

Este cálculo pode ser feito em um subprograma que recebe os números de entrada de alguma maneira e que devolva o valor correto também segundo uma convenção.

Esta convenção, em *Pascal*, é dada pelo protótipo de uma função que é constituído por três parâmetros: o nome da função, os números n e p envolvidos e, finalmente, o tipo do valor devolvido pelo subprograma, isto é, um tipo que contenha o número de vezes em que n é dividido por p.

```
function num_vezes_que_divide(p,n : integer): integer;
```

Outro trecho de código repetido é a atualização do MDC para receber a menor potência do primo sendo calculado, na primeira vez é o 2, nas outras vezes é um primo ímpar.

Basta criarmos um segundo protótipo de função que calcule a potência do primo elevado ao valor do menor contador. Este pode ser o seguinte:

```
function potencia(n,i : integer): integer;
```

Estas interfaces nos permitem modificar o programa do cálculo do MDC pela definição conforme mostra a figura 9.8.

Observamos que o uso de funções e procedimentos permite muita flexibilidade ao programador, pois ele pode alterar o código da funções, se quiser, tornando-as mais eficientes (caso não fossem) sem que haja efeito colateral no programa principal. As figuras 9.9 e 9.10 mostram sugestões de código para as funções.

Na verdade, este código não é muito apropriado, pois exige um comportamento não muito elegante na função num_vezes_que_divide, ela tem o *efeito colateral* de alterar o valor da variável n, o que não é natural dado o nome da função. Deveria apenas contar o número de vezes que divide e não fazer mais nada. O problema não pode ser resolvido com este mesmo algoritmo sem este efeito colateral. Em todo caso, sabemos que o algoritmo de Euclides é mais eficiente, mais simples de implementar e sobretudo mais elegante!

```
program mdcpeladefinicao; (* pela definicao de mdc *)
var
   a, b, primo, mdc: longint;
   cont_a, cont_b, menor_cont : integer;

function num_vezes_que_divide(p: integer; var n: longint): integer;
(* codigo da funcao num_vezes_que_divide *)

function potencia(n,p : integer): integer;
(* codigo da funcao potencia *)

begin (* programa principal *)
      read (a,b);
      mdc:= 1;

      cont_a:= num_vezes_que_divide(2,a);
      cont_b:= num_vezes_que_divide(2,b);

      if cont_a <= cont_b then
            menor_cont:= cont_a
      else
            menor_cont:= cont_b;

      mdc:= mdc * potencia(2,menor_cont);
      writeln ('mdc= ',mdc);

      primo:= 3;
      while (a <> 1) and (b <> 1) do
      begin

            writeln (primo);
            cont_a:= num_vezes_que_divide(primo,a);
            cont_b:= num_vezes_que_divide(primo,b);

            if cont_a <= cont_b then
                  menor_cont:= cont_a
            else
                  menor_cont:= cont_b;

            mdc:= mdc * potencia(primo,menor_cont);

            primo:= primo + 2;
      end;
      writeln (mdc);
end.
```

Figura 9.8: Calcula MDC entre a e b pela definição usando funções.

```
function num_vezes_que_divide(p: integer; var n: longint): integer;
var cont: integer;
begin
      (* descobre quantas vezes o 2 divide as duas entradas *)
      cont:= 0;
      while n mod p = 0 do
      begin
            cont:= cont + 1;
            n:= n div p;
      end;
      writeln ('num_vezes_que_divide= ',cont);
      num_vezes_que_divide:= cont;
end;
```

Figura 9.9: Calcula quantas vezes um número divide outro.

```
function potencia(n,p : integer): integer;
var pot: longint;
    i: integer;
begin
      pot:= 1;
      i:= 1;
      while i <= p do
      begin
            pot:= pot * n;
            i:= i + 1;
      end;
      writeln ('potencia= ',pot);
      potencia:= pot;
end;
```

Figura 9.10: Calcula a potência de um número elevado a outro.

9.4.1 Calculando raízes de equações do segundo grau

Para reforçarmos os conceitos em estudo, consideremos aqui o problema de se ler os coeficientes a, b e c que definem uma equação do segundo grau $ax^2 + bx + c = 0$ e imprimir as raízes calculadas pela fórmula de Bhaskara. O programa deve imprimir mensagens corretas no caso de não haverem raízes reais bem como não deve aceitar entradas cujo valor para o coeficiente a sejam nulas. O programa da figura 9.11 contém o código que resolve este problema.

A figura 9.12 ilustra o programa principal modificado para se dar a ideia de que as funções se comportam como expressões aritméticas, ao contrário dos procedimentos, que se comportam como comandos.

Na verdade, o programa principal poderia ser apenas o código da figura 9.13, sem prejuízo do funcinamento, mas com bem menos legilidade e ainda por cima sem tratamento do delta negativo. Apresentamos esta versão apenas para ilustrar o uso das funções dentro de funções, mas observamos que o cálculo do delta é feito duas vezes.

```
program bhaskara_v2;
var a, b, c, delta, x1, x2: real;

procedure ler (var a, b, c: real);
begin
      {$i-}
      repeat
            read (a, b, c);
      until (ioresult = 0) and (a <> 0);
      {$i+}
end;

function calcula_delta (a, b, c: real): real;
begin
      calcula_delta:= b*b - 4*a*c;
end;

function menor_raiz (a, b, delta: real): real;
begin
      menor_raiz:= (-b - sqrt(delta))/(2*a);
end;

function maior_raiz (a, b, delta: real): real;
begin
      maior_raiz:= (-b + sqrt(delta))/(2*a);
end;

begin (* programa principal *)
      ler (a, b, c); (* garante-se que a nao eh nulo *)
      delta:= calcula_delta (a, b, c);
      if delta >= 0 then
      begin
            x1:= menor_raiz (a, b, delta);
            writeln (x1);
            x2:= maior_raiz (a, b, delta);
            writeln (x2);
      end
      else
            writeln ('raizes complexas');

end.
```

Figura 9.11: Calculando raízes de equação do segundo grau.

```
begin (* programa principal *)
      ler (a, b, c); (* garante-se que a nao eh nulo *)
      delta:= calcula_delta (a, b, c);
      if delta >= 0 then
      begin
            writeln (menor_raiz (a, b, delta), maior_raiz (a, b, delta));
      end
      else
            writeln ('raizes complexas');

end.
```

Figura 9.12: Calculando raízes de equação do segundo grau.

```
begin (* programa principal *)
      ler (a, b, c); (* garante-se que a nao eh nulo *)
      writeln (menor_raiz (a, b, calcula_delta(a,b,c)),
               maior_raiz (a, b, calcula_delta(a,b,c)));
end.
```

Figura 9.13: Calculando raízes de equação do segundo grau.

Terminamos aqui a primeira parte do curso, no qual as noções fundamentais sobre algoritmos estão estabelecidas. Nos próximos estudaremos as principais estruturas de dados básicas para um curso introdutório de algoritmos.

9.5 Exercícios

1. Fazer uma função em *Pascal* que receba como parâmetro dois números inteiros não nulos e retorne TRUE se um for o contrário do outro e FALSE em caso contrário. Isto é, se os parâmetros forem 123 (cento e vinte e três) e 321 (trezentos e vinte e um), deve-se retornar TRUE. Usar apenas operações sobre inteiros.

2. Fazer uma função em *Pascal* que receba como parâmetro um número inteiro representando um número binário e retorne seu valor equivalente em decimal. Por exemplo, se a entrada for 10001, a saída deve ser 17.

3. Fazer uma função em *Pascal* que receba como parâmetro um número inteiro e retorne TRUE se ele for primo e FALSE em caso contrário. Use esta função para imprimir todos os números primos entre 0 e 1000.

4. Implemente funções para seno e cosseno conforme definidos em capítulos anteriores e use-as em uma terceira função que calcule a tangente. O programa principal deve imprimir os valores de $tg(x)$ para um certo valor fornecido pelo usuário.

5. Implemente um procedimento para gerar mais de um milhão de números inteiros. Os números gerados deverão ser impressos em um arquivo texto.

6. Faça uma função em *Pascal* que some dois números representando horas. A entrada deve ser feita da seguinte maneira:
   12 52
   7 13
   A saída deve ser assim:
   12:52 + 7:13 = 20:05

7. Faça uma função que receba como parâmetros seis variáveis DIA1, MES1 e ANO1, DIA2, MES2 e ANO2, todas do tipo integer. Considerando que cada trinca de dia, mês e ano representa uma data, a função deve retornar true se a primeira data for anterior à segunda e false caso contrário.

Capítulo 10

Estruturas de dados

Até aqui apresentamos as técnicas básicas para construção de algoritmos, incluindo as noções de funções e procedimentos. Podemos dizer que é possível, com este conteúdo, programar uma vasta coleção de algoritmos, inclusive alguns com alta complexidade.

Contudo, o estudo geral da disciplina de "Algoritmos e Estrurutas de Dados" envolve algoritmos que trabalham dados organizados em memória de maneira mais sofisticada do que as simples variáveis básicas que foram estudadas até o momento. É algo mais ou menos parecido como manter um guarda-roupas organizado no lugar de um monte de coisas atiradas no meio do quarto de qualquer jeito.

A organização de dados em memória permite a construção de algoritmos sofisticados e eficientes. Neste textp estudaremos três estruturas de dados elementares. São elas:

- vetores (ou array unidimencional);
- matrizes (ou array multidimencional);
- registros.

Nas seções seguintes explicaremos cada uma delas, sempre motivados por problemas que exigem seu uso ou que facilitam a implementação. Também veremos nos próximos capítulos algumas estruturas que misturam estas três.

10.1 Vetores

Para motivar, vamos considerar o problema seguinte: ler uma certa quantidade de valores inteiros e os imprimir *na ordem inversa* da leitura. Isto é, se os dados de entrada forem: 2, 5, 3, 4, 9, queremos imprimir na saída: 9, 4, 3, 5, 2.

Este tipo de problema é impossível de ser resolvido com o uso de apenas uma variável pois, quando se lê o segundo número, já se perdeu o primeiro da memória. Exigiria o uso de tantas variáveis quantos fossem os dados de entrada, mas notem que isto deve ser conhecido *em tempo de compilação*! O que faz com que simplesmente não seja possível resolver este problema para uma quantidade indefinida de valores.

De fato, quando se aloca, por exemplo, um número inteiro em uma variável de nome a, o que ocorre é que o computador reserva uma posição de memória em algum endereço da RAM (conforme sabemos pelo modelo Von Neumann). Um inteiro exige (dependendo da implementação) 2 bytes.

Mas, digamos que é preciso alocar espaço para 100 números inteiros. Sabendo que cada um deles ocupa 2 bytes, precisaríamos encontrar uma maneira de reservar $100 \times 2 = 200$ bytes e fazer com que este espaço de memória pudesse ser acessado também por um único endereço, ou em outras palavras, por uma única variável.

Os vetores são estruturas de dados que permitem o acesso a uma grande quantidade de dados em memória *usando-se apenas um nome de variável*. Esta variável especial é declarada de tal maneira que o programador passa a ter acesso à muitas posições de memória, de maneira controlada.

Sem ainda entrar em detalhes da linguagem *Pascal*, vamos tentar entender o processo.

Como funciona isto em memória?

Seguindo no exemplo de se alocar 200 espaços em memória para números inteiros, suponhamos que o seguinte comando faz esta alocação:

```
var V: array[1..200] of integer;
```

Em *Pascal* isto resulta na alocação de 200 vezes 2 bytes. Pela variável V temos o controle deste espaço.

O problema é *como* se faz para se escrever um valor qualquer neste espaço. Outro problema é *onde* se escreve, já que temos 200 possibilidades de escolha. O simples uso da variável, como estávamos acostumados, não serve. É preciso uma outra informação adicional para se dizer *em qual das 200 posições se quer escrever*.

Na verdade, a variável V aponta para o início do segmento reservado, da mesma maneira que se fazia para variáveis básicas já estudadas. Para se escrever em algum lugar deste segmento, é preciso informar, além do nome da variável, uma segunda informação: a posição (ou o deslocamento) dentro do espaço reservado.

Ora, sabemos que foram reservadas 200 posições, cada uma delas com espaço para conter um número inteiro. Se quisermos escrever na quinta posição, basta informar ao computador que o início do segmento é dado pela variável V e que, antes de se escrever, é preciso realizar um deslocamento de 5 posições, cada uma delas para um inteiro. Isto dá um deslocamento de 10 bytes. Após esta informação, o valor pode ser escrito. Se o desejo é escrever na décima quarta posição, o deslocamento deve ser de 14×2 bytes, isto é, 28 bytes.

Para se recuperar a informação, por exemplo para se imprimir, ou para se fazer algum cálculo com estes dados, basta usar o mesmo processo: os dados são acessados pelo nome da variável e pelo deslocamento.

Este processo foi apresentado em muito baixo nível. Como de costume, precisamos de uma outra forma de representação de mais alto nível. Isto é, cada linguagem de

programação que implementa a noção de vetores tem que encontrar uma maneira para se mascarar para o programador este processo que é baseado em deslocamentos (ou somas de endereços).

Na próxima seção veremos como a linguagem *Pascal* lida com isto.

Vetores em *Pascal*

Para se declarar um vetor de 200 posições inteiras, a linguagem *Pascal* usa a seguinte sintaxe (lembre-se que em outras linguagens a sintaxe pode ser diferente):

```
var v: array [1..200] of integer;
```

A construção "1..200" indica que existem 200 posições controladas pela variável v. O "of integer" indica que cada posição é para se guardar um número inteiro, isto é, 2 bytes (dependendo da implementação).

A rigor, a linguagem *Pascal* permite que se reserve 200 posições de várias maneiras. Basta que o intervalo "a..b" contenha 200 posições. Apresentamos 6 variantes dentre as milhares possíveis:

```
var v: array [0..199] of integer;

var v: array [201..400] of integer;

var v: array [-199..0] of integer;

var v: array [-300..-99] of integer;

var v: array [-99..100] of integer;

const min=11, max=210;
var v: array [min..max] of integer;
```

Em todas estas variantes, o intervalo define 200 posições. Em termos gerais, existe uma restrição forte. O intervalo deve ser definido em termos de números de algum tipo *ordinal* (em *Pascal*), isto é, *integer*, *logint*, ..., até mesmo *char*. Também em *Pascal*, o limite inferior deve ser menor ou igual ao limite superior. Na sequência desta seção, vamos considerar a versão que usa o intervalo de 1 a 200.

Agora, para guardar um valor qualquer, digamos 12345, na posição 98 do vetor v, em *Pascal*, se usa um dos dois comandos seguintes:

```
v[98]:= 12345;

read(v[98]); (* e se digita 12345 no teclado *)
```

Em termos gerais, vejamos os seguintes exemplos, para fixar o conceito:

```
read (v[1]); (* le do teclado e armazena na primeira posicao de v *)

i:= 10;
v[i+3]:= i * i; (* armazena o quadrado de i (100) na posicao 13 de v *)

write (i, v[i]); (* imprime o par (10, 100) na tela *)

write (v[v[13]]); (* imprime o valor de v[100] na tela *)

v[201]:= 5; (* gera erro, pois a posicao 201 nao existe em v *)

v[47]:= sqrt (4); (* gera erro, pois sqrt retorna um real, mas v eh de inteiros *)

var x: real;
v[x]:= 10; (* gera erro, pois x eh do tipo real, deveria ser ordinal *)
```

Note que a construção ($v[v[13]]$) só é possível pois o vetor v é do tipo *integer*. Se fosse um vetor de reais isto não seria possível (em *Pascal*).

10.1.1 Primeiros problemas com vetores

Para iniciarmos nossa saga pelos algoritmos sofisticados que usam vetores, vamos apresentar uma série de problemas já conhecidos para vermos como eles podem ser resolvidos usando vetores. Aproveitaremos para fixar conceitos já ensinados sobre procedimentos e funções. Desta forma o estudante poderá, resolvendo exercícios simples, se concentrar na novidade, isto é, no uso de vetores.

Lendo vetores

Para resolvermos o problema apontado acima, isto é, um programa que leia 10 números reais e os imprima na ordem inversa da leitura, precisamos inicialmente ler os elementos do vetor. O código da figura 10.1 ilustra uma solução possível. Quando executado, considerando que no teclado foi digitada a sequência 15, 12, 27, 23, 7, 2, 0, 18, 19 e 21, teremos em memória algo como ilustrado na figura seguinte:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

É importante, neste momento, atentar para alguns fatos:

1. este vetor tem 200 elementos, pois seu tamanho foi definido em tempo de compilação. Como só foram lidos os 10 primeiros elementos, o restante do vetor contém valores indefinidos que já estavam em memória quando programa começou a executar. Isto é o que chamamos de *lixo* de memória e está representado com interrogações na figura acima;

```
program lendo_vetores;
var v: array [1..200] of real;   (* define um vetor de reais *)
    i: integer;

begin
     i:= 1;
     while i <= 10 do
     begin
          read (v[i]);
          i:= i + 1;
     end;
end.
```

Figura 10.1: Lendo elementos e colocando no vetor.

2. o enunciado não especifica onde se armazenar os valores, como o vetor v tem 200 posições, poderíamos ter usado qualquer intervalo, mas normalmente se usa um vetor a partir da posição 1, e os algoritmos não podem deixar "buracos" no vetor;

3. o uso da variável auxiliar i no programa facilita muito a manipulação de vetores. Senão teríamos que usar comandos do tipo: *read*(v[1]), *read*(v[2]), *read*(v[3]), ..., recaindo nos problemas do início do curso;

4. a título de exemplo, mostramos a versão deste programa usando os comando *repeat* e *for*. Os trechos de código duas figuras 10.2 e 10.3, abaixo, ilustram estas duas variantes.

```
begin
     for i:= 1 to 10 do
          read (v[i]);
end.
```

Figura 10.2: Lendo elementos e colocando no vetor, usando *for*.

```
begin
     i:= 1;
     repeat
          read (v[i]);
          i:= i + 1;
     until i > 10;
end.
```

Figura 10.3: Lendo elementos e colocando no vetor, usando *repeat*.

Uma vez que se leu o vetor, pode-se agora manipular os dados da maneira necessária para se resolver o problema desejado. Nas seções seguintes vamos exemplificar usando diversos algoritmos já conhecidos.

Imprimindo vetores

O programa ilustrado na figura 10.4 mostra como ler 10 números reais do teclado e imprimí-los na tela. Usaremos o comando *for* nos exemplos seguintes pois ele facilita muito a redação dos programas que manipulam vetores. Os códigos ficam mais compactos e legíveis.

```
program lendo_e_imprimindo_vetores;
var v: array [1..200] of real;
    i: integer;

begin
     for i:= 1 to 10 do
     begin
          read (v[i]);
          writeln (v[i]);
     end;
end.
```

Figura 10.4: Lendo e imprimindo usando vetores.

É importante observar que este programa poderia ter sido resolvido sem o uso de vetores, como mostrado na figura 10.5.

```
program lendo_e_imprimindo;
var x: real; i: integer;

begin
     for i:= 1 to 10 do
     begin
          read (x);
          writeln (x);
     end;
end.
```

Figura 10.5: Lendo e imprimindo sem usar vetores.

Mostramos esta versão para o leitor poder comparar os dois códigos e perceber que a principal diferença entre as duas soluções é que, na versão com vetores, todos os números lidos do teclado continuam em memória após a leitura do último número digitado, enquanto que na versão sem vetores, a variável x teria armazenado apenas e tão somente o último número lido.

Uma outra maneira de escrever código para resolver o mesmo problema é separar o programa em duas partes: uma para a leitura e a outra para a impressão. O resultado é o mesmo, apenas a maneira de fazer difere. A figura 10.6 ilustra esta solução.

```
program lendo_e_imprimindo_vetores;
var v: array [1..200] of real;
    i: integer;

begin
     (* este pedaco de codigo trata apenas da leitura dos dados *)
     for i:= 1 to 10 do
          read (v[i]);

     (* este outro pedaco de codigo trata apenas da impressao   *)
     for i:= 1 to 10 do
          writeln (v[i]);
end.
```

Figura 10.6: Lendo e imprimindo: outra versão.

O código apresenta uma certa modularidade, pois pode ser facilmente visualizado como contendo uma parte que se ocupa da leitura dos dados separada da outra parte que se ocupa da impressão dos dados.

Apesar do fato deste programa funcionar, insistimos que ele merece ser escrito seguindo as boas técnicas de programação. Neste sentido o uso de funções e procedimentos pode ser explorado para que os dois módulos do programa (leitura e impressão) possam ficar claramente destacados de maneira independente um do outro.

O resultado final é o mesmo em termos de execução do programa e de seu resultado final na tela, exibido para quem executa o programa. Por outro lado, para o programador, o código é mais elegante. Por isto, vamos reescrever mais uma vez o programa, desta vez usando procedimentos.

Antes disto, porém, é importante destacar uma limitação da linguagem *Pascal*: infelizmente, o compilador não aceita um parâmetro do tipo *array*. Assim, a construção seguinte gera um erro de compilação:

```
procedure ler (var v: array [1..200] of real);
```

Para contornar este problema, a linguagem *Pascal* permite a definição de novos tipos de dados baseados em outros tipos pré-existentes. Isto se consegue com o uso da declaração *type*.

```
type vetor= array [1..200] of real;
```

O que ocorre com o uso da declaração *type* é que o nome do tipo *vetor* passa a ser conhecido pelo compilador, que por *default* só conhece os tipos pré-definidos da linguagem. O compilador *Pascal* foi um dos primeiros a permitir que o programador pudesse definir seus próprios tipos.

Assim, para reescrevermos o programa da figura 10.1 usando todo o nosso arsenal de conhecimentos adquiridos sobre procedimentos, funções, uso de constantes no código, comentários no código, ..., faríamos como apresentado na figura 10.7.

```
program ler_e_imprimir_com_procedures;
const min=1; max=200;
type vetor= array [min..max] of real;
var v: vetor;

procedure ler_vetor (var w: vetor);
var i: integer;
begin
     for i:= 1 to 10 do
          read (w[i]);
end;

procedure imprimir_vetor (var w: vetor); (* impressao dos dados *)
var i: integer;
begin
     for i:= 1 to 10 do
          write (w[i]);
end;

begin (* programa principal *)
     ler_vetor (v);
     imprimir_vetor (v);
end.
```

Figura 10.7: Lendo e imprimindo, agora com procedimentos.

Agora estamos prontos para resolver o problema proposto no início deste capítulo, aquele de ler uma sequência de números e imprimí-los ao contrário. Uma vez que os dados estão carregados em memória, após a execução do procedimento *ler(v)*, podemos manipular os dados da maneira que for necessário. No nosso caso, para imprimir ao contrário, basta modificar o procedimento de impressão para percorrer o vetor do final ao início. A figura 10.8 contém esta modificação. Basta agora modificar o programa principal trocando a chamada *imprimir(v)* por *imprimir_ao_contrario(v)*.

```
procedure imprimir_ao_contrario (var w: vetor);
var i: integer;
begin
     for i:= 10 downto 1 do
          write (w[i]);
end;
```

Figura 10.8: Procedimento que imprime os elementos do vetor ao contrário.

Algumas observações importantes:

1. A leitura é feita obrigatoriamente usando-se passagem de parâmetros por referência. A impressão pode usar passagem por valor. Contudo, conhecendo o fato de que existe uma cópia de elementos que é feita na pilha de memória, isto evidentemente provocará uma computação extra que pode custar caro, especialmente no caso em que os vetores são grandes. Imagine copiar, a toa, um milhão de elementos. Assim, em geral, vetores são passados sempre por referência.

2. O código seria generalizado facilmente se tivéssemos passado como parâmetro o tamanho usado (ou útil) do vetor, e não o número fixo 10, além do endereço dele. Neste caso, o código da figura 10.9 seria a solução mais elegante para o problema. Observar que o tamanho do vetor é lido dentro do procedimento, o que exige um parâmetro por referência.

```
program ler_e_imprimir_ao_contrario;
const min=0; max=50;
type vetor= array [min..max] of real;
var v: vetor;
    n: integer;

procedure ler (var w: vetor; var tam: integer); (* leitura *)
var i: integer;
begin
     read (tam);  (* 1<= tam<= 200, define o tamanho util do vetor *)
     for i:= 1 to tam do
          read (w[i]);
end;

procedure imprimir_ao_contrario (var w: vetor; tam: integer); (* impressao *)
var i: integer;
begin
     for i:= tam downto 1 do
          write (w[i]);
end;

begin (* programa principal *)
     ler (v, n);
     imprimir_ao_contrario (v, n);
end.
```

Figura 10.9: Lendo e imprimindo ao contrário, versão final.

Neste ponto esgotamos o assunto de ler e imprimir vetores e estamos prontos para novos problemas cujas soluções requerem o uso de vetores, ou tornam o código mais elegante.

Nas seções que seguem, vamos considerar dois vetores de tamanhos diferentes, um de inteiros o outro de reais. Nas apresentações dos algoritmos vamos omitir sempre que possível a redação dos cabeçalhos dos programas e nos concentrar apenas na solução dos novos problemas, sempre usando funções e procedimentos. As seguintes definições serão usadas até o final deste capítulo:

```
const min_r=0; max_r=50;
      min_i=1; max_i=10;

type vetor_r= array [min_r..max_r] of real;
     vetor_i= array [min_i..max_i] of integer;
```

Uma última observação, antes de continuarmos. Quando usamos vetores, estamos limitados ao tamanho dele, que deve ser conhecido *em tempo de compilação*! Isto pode causar dificuldades na resolução de problemas que envolvem um número desconhecido de valores de entrada. Mas não tem outro jeito a não ser, em tempo de compilação, se estimar um valor máximo para o número de elementos no vetor e, durante a execução, testar se este valor nunca foi ultrapassado. Se o número for maior, então deve-se modificar o tamanho do vetor e recompilar.

A questão de qual o tamanho ideal para ser o escolhido na hora de compilar é questão de bom-senso e envolve saber de qual aplicação se está falando. Por exemplo, se for um vetor para armazenar jogos da mega-sena, então o número 10 é suficiente. Se for para guardar saldos de clientes do banco, melhor saber quantos clientes existem hoje e estimar uma margem de erro que depende também do crescimento médio do número de clientes nos últimos anos. E assim por diante.

Imprimindo os que são pares

Vamos retornar ao velho e conhecido problema de se ler uma massa de dados de quantidade indefinida e imprimir apenas aqueles que são pares.

O programa da figura 10.10 ilustra uma procedure com uma possível solução. A leitura dos dados é muito similar ao que já foi mostrado no exemplo anterior, basta adaptar o tipo e dados vetor de reais para vetor de inteiros e por isto apresentamos apenas o que é diferente. Observemos a similaridade deste programa com relação ao código apresentado na figura 9.3.

```
procedure imprimir_pares (var v: vetor_i; tam: integer);
var i: integer;
begin
     for i:= 1 to tam do
          if eh_par (v[i]) then
               write (v[i],' ');
     writeln;
end;
```

Figura 10.10: Imprimindo os elementos do vetor que são pares.

Aqui se faz uso da função booleana "eh_par", que foi estudada na seção 9.2.5. Com isto concluímos que os problemas são os mesmos, apenas o uso deles é ligeiramente diferente por dois motivos: usa-se funções ou procedimentos, e também se resolve usando-se vetores. O resto não muda.

Um problema que pode parecer o mesmo, mas não é, seria imprimir os elementos das posições pares do vetor, e não mais os elementos cujos conteúdos são pares. Perceber esta diferença é fundamental no estudo de vetores. Consideremos o seguinte vetor v com tamanho 10 como exemplo:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

O código da figura 10.10 produziria como saída o seguinte: 12, 2, 0 e 18, que estão respectivamente nas posições 2, 6, 7 e 8 do vetor. Se, na linha 5 do programa, nós testarmos se o índice é par (e não o conteúdo):

```
if eh_par (i) then  // no lugar de: if eh_par (v[i]) then
```

então a saída seria: 12, 23, 2, 18 e 21, respectivamente para os elementos das posições 2, 4, 6, 8 e 10 do vetor. Oberve com atenção esta diferença, é importante.

Como antes, a função "eh_par" pode ser substituída por qualquer outra, como por exemplo, ser primo, se divisível por n, pertencer à sequência de Fibonacci, ou qualquer outra propriedade mais complexa que se queira.

Encontrando o menor de uma sequência de números

Vamos considerar o problema de se ler uma sequência de $N > 1$ números do teclado e imprimir o menor deles. Já sabemos resolvê-lo sem o uso de vetores, conforme ilustrado na figura 10.11.

```
program menor_dos_lidos;
var N, i: integer; x, menor: real;
begin
      read (N);
      read (x);
      menor:= x;
      for i:= 2 to N do
      begin
            read (x);
            if x < menor then
                  menor:= x;
      end;
      writeln (menor);
end.
```

Figura 10.11: Encontrando o menor de N números lidos, sem vetor.

Vamos reimplementá-lo em termos de uma função que considera que os elementos digitados no teclado já foram lidos em um vetor. A figura 10.12 permite percebermos que a técnica usada foi rigorosamente a mesma: o primeiro número lido é considerado o menor de todos. Na leitura dos dados subsequentes, quando um número ainda menor é encontrado, atualiza-se a informação de quem é o menor de todos.

```
function menor_dos_lidos (var v: vetor_r; N: integer): real;
var i: integer; menor: real;
begin
      menor:= v[1];
      for i:= 2 to N do
            if v[i] < menor then
                  menor:= v[i];
      menor_dos_lidos:= menor;
end;
```

Figura 10.12: Encontrando o menor de N números lidos, com vetor.

10.1.2 Soma e produto escalar de vetores

Nesta seção vamos explorar algoritmos ligeiramente mais sofisticados, envolvendo noções novas sobre vetores[1].

Somando vetores

Nesta seção vamos implementar o algoritmo que soma dois vetores. Para isto precisamos antes entender como funciona o processo.

Sejam v e w dois vetores. Para somá-los, é preciso que eles tenham o mesmo tamanho. Isto posto, o algoritmo cria um novo vetor $v + w$ no qual cada elemento $(v + w)[i]$ do novo vetor é a soma dos respectivos elementos $v[i]$ e $w[i]$. O esquema é conforme a figura seguinte, que apresenta dois vetores de 6 elementos cada:

	1	2	3	4	5	6
v:	2	6	1	5	3	3
	+	+	+	+	+	+
w:	1	3	2	4	3	5
	=	=	=	=	=	=
v+w:	3	9	3	9	6	8

Assim, o algoritmo que soma os dois vetores deverá, para cada i fixo, somar os respectivos elementos em v e w e guardar em $v + w$. Variar i de 1 até o tamanho do vetor resolve o problema. O programa da figura 10.13 implementa esta ideia.

[1] Não tão novas, já que são conceitos estudados no ensino médio.

```
procedure somar_vetores (var v, w, soma_v_w: vetor_i; tam: integer);
(* este procedimento considera que os dois vetores tem o mesmo tamanho *)
var i: integer;

begin
      for i:= 1 to tam do
            soma_v_w[i]:= v[i] + w[i];
end;
```

Figura 10.13: Somando dois vetores.

Importante notar que i é variável local, isto é, serve apenas para controlar o comando *for* interno do procedimento. Também digno de nota é a passagem de parâmetros: no caso de v e w, poderíamos perfeitamente ter passado por valor, pois não são alterados no procedimento. Isto vale também para o tamanho do vetor. Mas *soma_v_w* deve ser obrigatoriamente passado por referência.

Se, por acaso, os dois vetores tivessem eventualmente tamanhos diferentes o protótipo mudaria um pouco e poderia ser assim (desde que na implementação se teste se $tam\_v = tam\_w$):

```
procedure soma_vetores (var v: vetor_i; tam_v: integer;
                        var w: vetor_i; tam_w: integer;
                        var soma_v_w: vetor_i; tam_soma: integer);
```

Produto escalar

Nesta seção vamos implementar o algoritmo que calcula o produto escalar de dois vetores. Para isto precisamos antes entender como funciona o processo.

Sejam v e w dois vetores. Para se obter o produto escalar é preciso que eles tenham o mesmo tamanho. Isto posto, o algoritmo gera um número real (ou inteiro, depende do tipo de dados do vetor) obtido pela soma das multiplicações de cada elemento i dos vetores dados, respectivamente. O esquema é conforme a figura seguinte, que apresenta dois vetores de 6 elementos cada:

	1	2	3	4	5	6
v:	2	6	1	0	3	3
	×	×	×	×	×	×
w:	1	0	2	4	3	5
	=	=	=	=	=	=
	2	0	2	0	9	15

Os números obtidos a partir das multiplicações para todos os i fixos devem ser somados: $2 + 0 + 2 + 0 + 9 + 15 = 28$. Logo, 28 é o produto escalar de v e w.

Assim, o algoritmo que calcula o produto escalar de dois vetores deverá, para cada i fixo, multiplicar os respectivos elementos em v e w e usar a técnica do acumulador para armazenar as somas parciais. Variar i de 1 até o tamanho do vetor resolve o problema. O programa que implementa esta ideia é apresentado na figura 10.14.

```
function prod_escalar (var v, w: vetor_r; tam: integer): real;
var i: integer;
    soma: real;
begin
     soma:= 0;
     for i:= 1 to tam do
          soma:= soma + v[i] * w[i];
     prod_escalar:= soma;
end;
```

Figura 10.14: Produto escalar de dois vetores.

Como procuramos mostrar, programar usando vetores, funções e procedimentos não é muito diferente de se programar os algoritmos elementares tais como os que foram estudados até então. Pelo menos até agora. A próxima seção vai apresentar novas técnicas usando-se estes novos conceitos.

10.1.3 Busca em vetores

Nesta seção vamos estudar alguns algoritmos para o importante problema de busca em vetores. Em outras palavras, estamos interessados em saber se um dado elemento x é um dos elementos de um vetor v e, caso seja, também podemos querer saber a posição onde este elemento se encontra.

Este problema é extremamente importante em computação e não é difícil imaginar aplicações no nosso dia a dia. Por exemplo, localizar um livro na biblioteca, localizar um filme na videolocadora, saber se um dado CPF está cadastrado em alguma lista de cheques, e por ai vai.

O tema de busca em vetores é tão importante que é estudado de diversas maneiras ao longo de um curso de ciência da computação. Em um curso introdutório iremos estudar os mais elementares para o caso de vetores de reais ou inteiros.

Ao longo desta seção, vamos considerar sempre que um dado vetor v já foi lido em memória de alguma maneira. Assim, vamos elaborar algoritmos na forma de funções ou procedimentos. A ideia é, como tem sido até aqui, inicar por algo trivial e evoluir a qualidade da solução.

O algoritmo mais ingênuo possível é mostrado na figura 10.15.

Este algoritmo sempre acha o elemento x em v se ele estiver presente, pois ele varre todo o vetor comparando cada elemento com x. Caso x esteja presente duas vezes ou mais, ele retorna a posição da última ocorrência. O algoritmo sempre faz n

```
function busca_simples (x: real; var v: vetor_r; n: integer): integer;
var i: integer;
begin
      busca_simples:= 0;
      for i:= 1 to n do
            if v[i] = x then
                  busca_simples:= i;
end;
```

Figura 10.15: Busca em vetores, primeira versão.

comparações, por causa do comando *for*. Por isto diz-se que o número de comparações que o algoritmo realiza é proporcional ao tamanho do vetor.

De fato, em qualquer programa sempre é interessante analisar o que custa mais caro no código. Entendemos por "custar caro" como sendo o comando que é executado o maior número de vezes durante uma rodada do programa.

Neste caso, temos um comando *if*, que faz um teste de igualdade (uma comparação), dentro do escopo de um comando *for*, o qual é controlado pela variável n. Logo, esta comparação será executada sempre n vezes.

Apesar do algoritmo funcionar, ele faz comparações demais: mesmo quando o elemento já foi encontrado o vetor é percorrido até o final. Obviamente ele poderia parar na primeira ocorrência, pois não foi exigido no enunciado que fossem localizadas todas as ocorrências (este é outro problema). Então podemos modificar o código para a versão apresentada na figura 10.16, simplesmente trocando-se o *for* por um *while* que termina a execução tão logo o elemento seja encontrado (se for encontrado).

```
function busca_simples_v2 (x: real; var v: vetor_r; n: integer): integer;
var i: integer;
    achou: boolean;

begin
      achou:= false;
      i:= 1;
      while (i <= n) and not achou do
      begin
            if v[i] = x then
                  achou:= true;
            i:= i + 1;
      end;
      if achou then
            busca_simples_v2:= i - 1;
end;
```

Figura 10.16: Busca em vetores, segunda versão.

Este algoritmo é mais rápido, na média, do que o anterior, embora o número de comparações feitas possa ser a mesma no caso do elemento não estar presente no

vetor. Mas, se dermos sorte, o elemento pode estar no início do vetor e terminar bem rápido. Na média, espera-se que ele pare mais ou menos na metade do vetor, isto é, considerando-se uma distribuição uniforme dos elementos.

Mas como o laço faz dois testes, no caso do elemento não ser encontrado ele será um pouco mais lento. Notem que o duplo teste no laço é necessário pois deve parar *ou* porque achou o elemento *ou* porque o vetor terminou. Este segundo teste só vai dar *true* uma única vez, o que é um despedício.

Se pudesse haver garantia de que sempre o elemento procurado estivesse presente, então poderíamos construir um teste simples, o que pode nos economizar alguma computação. Esta garantia não pode existir, certo? Mais ou menos. Digamos que o programador deliberadamente coloque o elemento no vetor. Neste caso, há a garantia de que ele está presente. Mas alguém pode perguntar: assim não vale, se eu coloco não significa que ele já estava presente, eu vou sempre encontrar o elemento "falso".

Novamente, depende de *onde* o programador coloque o elemento. Digamos que ele o coloque logo após a última posição. Então, das duas uma: ou o elemento não estava no vetor original, e neste caso a busca pode terminar pois o elemento será encontrado após a última posição; ou o elemento estava presente e será encontrado antes daquele que foi adicionado. Um teste final resolve a dúvida. Se for encontrado em posição válida, é porque estava presente, senão, não estava.

Este elemento adicionado logo após o final do vetor é denominado *sentinela* e seu uso é ilustrado na versão apresentada na figura 10.17.

```
function busca_com_sentinela (x: real; var v: vetor_r; n: integer): integer;
var i: integer;

begin
      v[n+1]:= x;
      i:= 1;
      while v[i] <> x do
            i:= i + 1;
      if i <= n then
            busca_com_sentinela:= i
      else
            busca_com_sentinela:= 0;
end;
```

Figura 10.17: Busca em vetores com sentinela.

Apesar da melhoria, este algoritmo sempre faz um número de comparações que pode atingir n no pior caso, isto é, quando o elemento não está presente no vetor.

O caso ótimo e o caso médio não mudam, embora o algoritmo, conforme explicado, faça metade das comparações quando comparado com a versão anterior. Desta forma ele é apenas ligeiramente melhor do que o anterior.

Ainda, o programador deve garantir que a posição usada pelo sentinela nunca seja usada como sendo um elemento válido, pois não é. O programador colocou o elemento ali de maneira controlada, mas não se trata de um elemento válido. Isto significa que o número de elementos úteis do vetor agora não é mais max_r (ou max_i), mas sempre

um a menos. Normalmente se modifica a definição do tipo para prever este espaço adicional para o programador.

```
const min_r=0; max_r=50;
      min_i=1; max_i=10;

type vetor_r= array [min_r..max_r+1] of real;
     vetor_i= array [min_i..max_i+1] of integer;
```

É possível tornar o algoritmo mais eficiente?

A resposta é sim[2]. Mas, para tornar a busca mais eficiente é preciso impor algumas restrições extra na forma como os dados são organizados em memória.

Esta maneira consiste em considerar que os dados estão de alguma forma respeitando uma ordem lexicográfica em memória. Por exemplo, se forem nomes, estão em ordem alfabética. Se forem números, estão em ordem crescente (ou descrescente). Porque isto é necessário? Pois pode-se explorar a informação de ordenação para tornar o método mais eficiente.

No caso, podemos modificar a solução anterior para que o algoritmo termine a busca sempre que encontrar um elemento que já é maior do que aquele que se está procurando, pois o vetor está ordenado. O programa da figura 10.18 foi implementado com esta ideia.

```
function busca_vetor_ordenado (x: real; var v: vetor_r; n: integer): integer;
var i: integer;

begin
     v[n+1]:= x;
     i:= 1;
     while v[i] < x do
          i:= i + 1;
     if (v[i] = x) and (i <= n) then
          busca_vetor_ordenado:= i
     else
          busca_vetor_ordenado:= 0;
end;
```

Figura 10.18: Busca em vetores ordenados.

Apesar de termos explorado uma propriedade adicional que faz com que no caso médio a busca seja mais eficiente, no pior caso, aquele em que o elemento procurado não pertence ao vetor, ainda temos um algoritmo que faz tantas comparações quanto o tamanho do vetor. É possível fazer melhor? A resposta novamente é sim.

Como exemplo, considere o problema de se encontrar um verbete no dicionário. Sabemos que estes verbetes se encontram em ordem alfabética. Por este motivo,

[2]Em disciplinas avançadas de algoritmos se estudam métodos bastante eficientes para este problema. No nosso caso veremos apenas um deles.

ninguém em sã consciência procura um nome em ordem sequencial desde o início, a menos que esteja procurando algo que começa com a letra "A".

Suponha que o verbete inicia com a letra "J". Normalmente se abre o dicionário mais ou menos no meio. Se o nome presente no iníco da página for "menor" lexicograficamente falando do que aquele que se busca, por exemplo, algo começando com a letra "D", então, pode-se tranquilamente rasgar o dicionário, eliminando-se tudo o que está antes do "D" e continuar o processo com o que restou.

Na segunda tentativa abre-se novamente mais ou menos na metade do que sobrou. Suponha que caímos na letra "M". Como estamos procurando o "J", pode-se sem problemas rasgar novamente o dicionário eliminando-se tudo o que segue o "M", até o fim. Resta procurar "apenas" entre o "D" e o "M".

Aplicando-se consistentemente este processo repetidas vezes, sempre teremos um dicionário dividido mais ou menos por 2. Se ele tinha 500 páginas no início, na segunda vez terá 250 e na terceira 125, na quarta algo próximo de 70 páginas. E assim por diante. Por isto que costumamos localizar rapidamente verbetes em um dicionário.

O algoritmo mostrado na figura 10.20, denominado de *busca binária*, implementa esta ideia. Procura-se o elemento no meio do vetor. Se encontrou, então pode parar. Se não encontrou, basta verificar se o valor encontrado é maior ou menor do que o procurado. Se for maior, joga-se a metade superior fora e trabalha-se apenas com a metade inferior. Se for menor, joga-se fora a metade inferior e trabalha-se apenas com a metade superior. Novamente procura-se na metade do que sobrou e assim por diante, até que se encontre o elemento ou até que se determine que não é mais possível encontrá-lo.

Vamos agora comparar estas últimas quatro versões para o algoritmo de busca. A tabela 10.19 resume o número de comparações feitas para diversos tamanhos de entrada, sempre analisando o pior caso. O caso médio exige um pouco mais de cuidado para se calcular e não será estudado aqui. O caso ótimo é sempre uma única comparação para os casos: o algoritmo acha o elemento na primeira tentativa.

Versão	$n = 10$	$n = 10^2$	$n = 10^4$	$n = 10^8$
Busca simples (fig. 10.16)	20	200	20000	200000000
Busca com sentinela (fig. 10.17)	10	100	10000	100000000
Busca em vetor ordenado (fig. 10.18)	10	100	10000	100000000
Busca binária (fig. 10.20)	3	5	10	19

Figura 10.19: Tabela resumindo número de comparações para algoritmos de busca.

O importante do método da busca binária é que ela apresenta um caráter *logarítmico* para o pior caso com relação ao tamanho da entrada, o que é bastante significativo. Contudo, é absolutamente relevante destacar que este método só pode ser aplicado em vetores ordenados, senão não funciona. A questão é saber qual o custo de se ordenar, ou de se manter ordenado, um vetor. Isto será estudado à frente.

```
function busca_binaria (x: real; var v: vetor_r; n: integer): integer;
var inicio, fim, meio: integer;

begin
     inicio:=1;                        (* aponta para o inicio do vetor *)
     fim:= n;                          (* aponta para o fim do vetor    *)
     meio:= (inicio + fim) div 2;      (* aponta para o meio do vetor   *)
     while (v[meio] <> x) and (fim >= inicio) do
     begin
          if v[meio] > x then     (* vai jogar uma das duas metades fora  *)
               fim:= meio - 1     (* metade superior foi jogada fora      *)
          else
               inicio:= meio + 1;(* metade inferior foi jogada fora       *)
          meio:= (inicio + fim) div 2;  (* recalcula apontador para meio *)
     end;
     (* o laco termina quando achou ou quando o fim ficou antes do inicio *)
     if v[meio] = x then
          busca_binaria:= meio
     else
          busca_binaria:= 0;
end;
```

Figura 10.20: Busca binária.

Manipulando vetores ordenados

Quando operações de inserção e remoção são feitas em vetores ordenados é importante que estas alterações em seus elementos preservem a propriedade do vetor estar ordenado, por exemplo, para que se possa usar uma busca binária.

A seguir apresentamos soluções para se remover ou inserir um elemento de um vetor, iniciando pela remoção. O procedimento da figura 10.21 garante a remoção do elemento que está na posição *pos* do vetor v que tem tamanho n.

```
procedure remove_vetor_ordenado (pos: integer; var v: vetor_r; var n: integer);
var i: integer;

begin
     for i:= pos to n-1 do
          v[i]:= v[i+1];
     n:= n - 1;
end;
```

Figura 10.21: Removendo de um vetor ordenado.

Remover elementos não é tão trivial quanto parece, pois não podemos deixar "buracos" no vetor. Este algoritmo então copia todos os elementos que estão à frente daquele que foi removido uma posição para trás. Tomemos como exemplo o seguinte vetor ordenado:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	19	21	23	27	?	?	?	?	?	...	?	?	?	?

Para remover o elemento 12, que está na posição 4, todos os outros são copiados, um por um. Para não perdermos elementos, este processo tem que iniciar da posição onde houve a remoção até a penúltima posição. O vetor resultante é o seguinte:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	15	18	19	21	23	27	27	?	?	?	?	?	...	?	?	?	?

O detalhe é que o elemento 27 que estava na posição 10 continua lá, mas, como o vetor agora tem tamanho 9 (pois um elemento foi removido) este último 27 agora é lixo de memória.

Com relação ao algoritmo de inserção de um elemento em um vetor ordenado, deve-se primeiro determinar a posição correta de inserção, logo, um processo de busca deve ser feito, iniciando do começo do vetor (posição 1) até que seja encontrado um elemento maior que aquele que está se inserindo. Para facilitar esta busca, usamos uma sentinela.

Tomemos novamente como exemplo o vetor abaixo e consideremos que vamos inserir o elemento 17.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	19	21	23	27	?	?	?	?	?	...	?	?	?	?

A posição correta do 17 é logo após o elemento 15, isto é, na posição 6. Para abrirmos espaço no vetor e ao mesmo tempo não perdermos o 18 que lá está devemos copiar todos os elementos, um por um, uma posição para frente. Isto deve ser feito de trás para frente, isto é, copia-se o último uma posição adiante para abrir espaço para colocar o penúltimo, e assim por diante. O resultado seria um vetor assim:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	18	19	21	23	27	?	?	?	?	...	?	?	?	?

Observe que agora temos duas vezes o 18 no vetor. O primeiro deles agora pode ser substituído pelo novo elemento, o 17, assim:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	17	18	19	21	23	27	?	?	?	?	...	?	?	?	?

O procedimento da figura 10.22 insere um elemento x no vetor v que tem tamanho n de maneira que ele continue ordenado. Apenas uma passagem no vetor é feita: ao mesmo tempo em que ele está procurando a posição correta do elemento ele já vai abrindo espaço no vetor.

```
procedure insere_vetor_ordenado (x: real; var v: vetor_r; var n: integer);
var i: integer;

begin
      v[0]:= x;
      i:= n;
      while x < v[i] do
      begin
            v[i+1]:= v[i];
            i:= i - 1;
      end;
      v[i+1]:= x;
      n:= n + 1;
end;
```

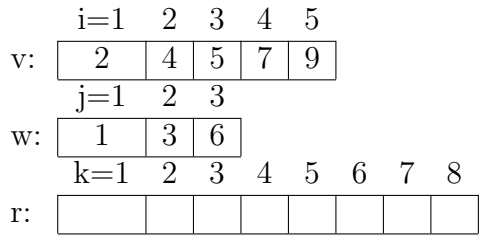
Figura 10.22: Inserindo em um vetor ordenado.

No caso de vetores não ordenados, os processos de inserção e remoção são mais simples. Pode-se sempre inserir um elemento novo no final do vetor e para remover, basta copiar o último sobre o que foi removido.

Há um último algoritmo interessante que trabalha com vetores ordenados: a ideia é fundir[3] dois vetores ordenados em um terceiro, obtendo-se um vetor ordenado como resultado contendo os mesmos elementos dos vetores anteriores.

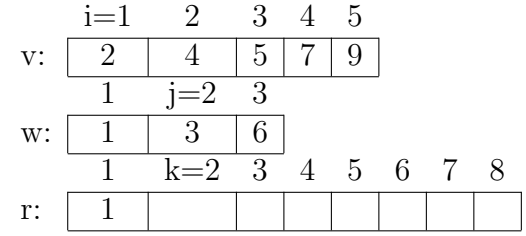
O princípio deste algoritmo é: atribui-se a duas variáveis i e j o índice 1. Em seguida se compara os respectivos elementos das posições i em um vetor e j no outro. Se o elemento do primeiro vetor é menor do que o do segundo, ele é copiado no terceiro vetor e o apontador i é incrementado de 1, passando a observar o próximo elemento. Senão, o elemento do segundo vetor é menor e portanto este que é copiado para o terceiro vetor e o apontador j que é incrementado. Quando um dos dois vetores acabar, basta copiar o restante do outro no terceiro vetor. Um apontador k controla o terceiro vetor. Este é sempre incrementado a cada cópia.

Vejamos isto através de um exemplo. Consideremos os seguintes dois vetores ordenados e um terceiro que vai receber a fusão dos dois.

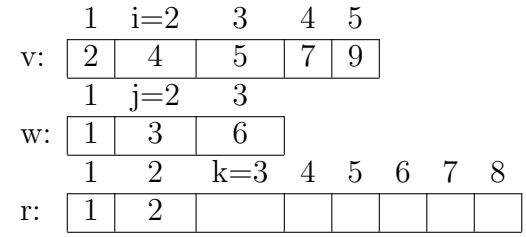


comparam-se o elementos apontados por i e j. Como $1 < 2$, copia-se o 1 no vetor r e incrementa-se o apontador j, assim:

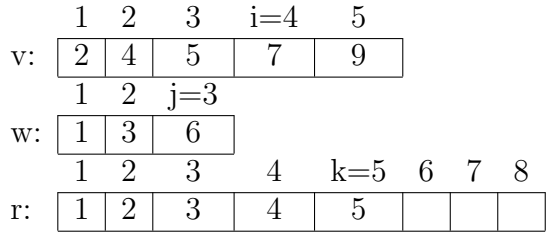
[3]Em inglês, *merge*



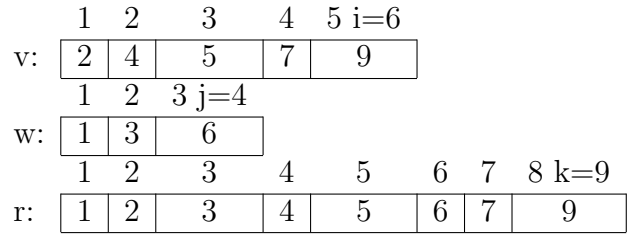
Agora comparam-se o elementos apontados por i e j. Como $2 < 3$, copia-se o 2 no vetor r e incrementa-se o apontador i, assim:



Repetindo-se este processo mais algumas vezes chegaremos no seguinte estado:



Neste ponto o 6 será copiado em r e o vetor w termina. Basta copiar o que sobrou de v e terminar o processo, o que resultará no seguinte:



O algoritmo da figura 10.23 implementa o que foi discutido, é eficiente e varre os dois vetores apenas uma vez.

```
procedure merge_vetores (var v: vetor_r; n: integer; var w: vetor_r; m: integer; var r:
    vetor_r);
var i, j, k, l: integer;

begin
    i:= 1; j:= 1; k:= 1;
    (* i vai controlar os elementos de v *)
    (* j vai controlar os elementos de w *)
    (* k vai controlar os elementos do vetor resultante da fusao, r *)

    while (i <= n) and (j <= m) do
    begin
        if v[i] <= w[j] then (* se o elemento de v dh menor que o de w *)
        begin
            r[k]:= v[i];
            i:= i + 1;
        end
        else                    (* o elemento de w eh menor que o de v *)
        begin
            r[k]:= w[j];
            j:= j + 1;
        end;
        k:= k + 1;              (* k eh sempre incrementado *)
    end;
    (* quando sai do laco, um dos dois vetores acabou *)

    if i <= n then (* w acabou, copiar o restante de v em r *)
        for l:= i to n do
        begin
            r[k]:= v[l];
            k:= k + 1;
        end
    else            (* v acabou, copiar o restante de w em r *)
        for l:= j to m do
        begin
            r[k]:= w[l];
            k:= k + 1;
        end;
end;
```

Figura 10.23: Fundindo dois vetores ordenados.

10.1.4 Ordenação em vetores

Ordenar vetores é extremamente importante em computação, pois é muito comum que uma saída de um programa seja dado com algum tipo de ordem sobre os dados. Nesta seção vamos apresentar dois algoritmos para este problema: os métodos da ordenação por seleção e por inserção.

Ordenação por seleção

A ordenação por seleção é um método bastante simples de se compreender e também de se implementar.

O princípio é selecionar os elementos corretos para cada posição do vetor, daí o nome do método. Para um vetor de N elementos, existem $N-1$ etapas. Em cada etapa i o i-ésimo menor elemento é selecionado e colocado na posição i.

Assim, na primeira etapa, o menor elemento de todos é localizado (selecionado) e colocado na primeira posição do vetor. Na segunda etapa localiza-se o segundo menor e coloca-se na segunda posição, e assim por diante, até que, por fim, o penúltimo menor elemento (isto é, o segundo maior) é colocado na penúltima posição. Consequentemente, como já temos $N-1$ elementos em seus devidos lugares, o último também está. Vejamos um exemplo de um vetor com 10 elementos.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

Para localizarmos o menor elemento, que é o zero que está na posição 7 do vetor, só há uma maneira, que é percorrer todo o vetor e localizar o menor. Este deve ser colocado na primeira posição. Este último (o 15), por sua vez, deve ser trocado de posição com o da posição 7. Por isto a busca pelo menor deve nos retornar o índice do menor elemento e não o elemento em sí. O resultado da primeira etapa deste processo está mostrado na figura abaixo, com destaque para os elementos trocados.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	12	27	23	7	2	**15**	18	19	21	?	?	?	?	?	...	?	?	?	?

Neste ponto precisamos do segundo menor. Por construção lógica do algoritmo, basta percorrer o vetor a partir da segunda posição, pois o primeiro já está em seu lugar. O menor de todos agora é o 2, que está na posição 6. Basta trocá-lo pelo segundo elemento, que é o 12. E o processo se repete:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	**2**	27	23	7	**12**	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	**7**	23	**27**	12	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	**12**	27	**23**	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	**15**	23	**27**	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	**18**	27	**23**	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	**19**	23	**27**	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	19	**21**	27	**23**	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	19	21	**23**	**27**	?	?	?	?	?	...	?	?	?	?

```
procedure selecao (var v: vetor_r; n: integer);
var i, j, pos_menor: integer; aux: real;

begin
     for i:= 1 to n–1 do
     begin
          (* acha a posicao do menor a partir de i *)
          pos_menor:= i;
          for j:= i+1 to n do (* inicia a partir de i+1 *)
               if v[j] < v[pos_menor] then
                    pos_menor:= j;

          aux:= v[pos_menor]; (* troca os elementos *)
          v[pos_menor]:= v[i];
          v[i]:= aux;
     end;
end;
```

Figura 10.24: Método de ordenação por seleção.

A figura 10.24 mostra a versão em *Pascal* deste algoritmo.

Este algoritmo tem algumas particularidades dignas de nota. A primeira é que, em cada etapa i, a ordenação dos primeiros $i-1$ elementos é definitiva, isto é, constitui a ordenação final destes primeiros i elementos do vetor.

A segunda é que a busca pelo menor elemento em cada etapa i exige percorrer um vetor de $N-i$ elementos. Como isto é feito N vezes, então o número de comparações feitas na parte mais interna do algoritmo é sempre $\frac{N(N-1)}{2}$, o que define um comportamento quadrático para as comparações.

A terceira é que trocas de posições no vetor ocorrem no laço mais externo, por isto o número total de trocas feitas pelo algoritmo é sempre $N-1$.

Ordenação por inserção

A ordenação por inserção é provavelmente a mais eficiente, na prática, que a ordenação por seleção. Porém, embora o algoritmo em sí seja simples, sua implementação é repleta de detalhes. Vamos inicialmente entender o processo.

O princípio do algoritmo é percorrer o vetor e a cada etapa i, o elemento da posição i é inserido (daí o nome do método) na sua posição correta *relativamente* quando comparado aos primeiros $i-1$ elementos.

Para melhor compreensão, faremos a apresentação de um exemplo sem mostrarmos o vetor, usaremos sequências de números. Consideremos que a entrada é a mesma do exemplo anterior, isto é:

15, 12, 27, 23, 7, 2, 0, 18, 19, 21.

Na primeira etapa o algoritmo considera que o primeiro elemento, o 15, está na sua posição relativa correta, pois se considera apenas a primeira posição do vetor. Usaremos os negritos para mostrar quais as etapas já foram feitas pelo algoritmo.

Na segunda etapa deve-se inserir o segundo elemento em sua posição relativa correta considerando-se apenas o vetor de tamanho 2. Como o 12 é menor que que o 15, deve-se trocar estes elementos de posição, nos resultando na sequência:

12, 15, 27, 23, 7, 2, 0, 18, 19, 21.

Neste ponto, os elementos 12 e 15 estão em suas posições relativas corretas considerando-se um vetor de 2 posições. Agora, deve-se colocar o 27 no vetor de 3 elementos. Mas o 27 já está em seu lugar relativo, então o algoritmo não faz nada:

12, 15, 27, 23, 7, 2, 0, 18, 19, 21.

Na quarta etapa deve-se inserir o 23 na sua posição relativa correta considerando-se um vetor de 4 elementos. O 23 tem que estar entre o 15 e o 27:

12, 15, 23, 27, 7, 2, 0, 18, 19, 21.

Na quinta etapa deve-se inserir o 7 na sua posição relativa a um vetor de 5 elementos. Ele deve ser inserido antes do 12, isto é, na primeira posição:

7, 12, 15, 23, 27, 2, 0, 18, 19, 21.

A situação para o 2 é similar, deve ser inserido antes do 7, isto é, no início:

2, 7, 12, 15, 23, 27, 0, 18, 19, 21.

Idem para o zero:

0, 2, 7, 12, 15, 23, 27, 18, 19, 21.

Agora é a vez de inserirmos o 18, entre o 15 e o 27:

0, 2, 7, 12, 15, 18, 23, 27, 19, 21.

Na penúltima etapa inserimos o 19 entre o 18 e o 23:

0, 2, 7, 12, 15, 18, 19, 23, 27, 21.

E por último o 21 entre o 19 e o 23:

0, 2, 7, 12, 15, 18, 19, 21, 23, 27.

Esta sequência de N passos é de fácil compreensão. Se fôssemos executar com um conjunto de cartas na mão, por exemplo, com cartas de baralho, imaginando um maço de cartas virado na mesa, basta pegar as cartas uma a uma e encaixar no lugar certo. As cartas de baralho são facilmente manipuladas para permitir uma inserção em qualquer posição.

Infelizmente esta operação executada em um vetor não é tão simples. Vamos considerar como exemplo a etapa 8 acima, isto é, inserção do 18 no lugar certo. Retomemos este caso agora considerando um vetor para melhor ilustração, com destaque para o elemento 18 que deve nesta etapa ser inserido no lugar certo:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	23	27	**18**	19	21	?	?	?	?	?	...	?	?	?	?

A posição correta do 18, como vimos, é entre o 15 e o 23, isto é, na sexta posição do vetor. Significa que os elementos das posições 6 e 7 devem ser movidos um para frente para abrir espaço no vetor para inserção do 18. Os elementos das posições 9 em diante não vão mudar de lugar. Executando esta operação, e salvando o 18 em alguma variável temporária, obteremos o seguinte vetor:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	23	**23**	**27**	19	21	?	?	?	?	?	...	?	?	?	?

Isto é, o 27 foi copiado da posição 7 para a posição 8 e o 23 foi copiado da posição 6 para a posição 7. Na figura acima destacamos os elementos que foram movidos de lugar. Observando que o 23 ficou repetido na posição 6, o que na prática resultou na posição 6 livre. Agora basta inserir o 18 aí:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	**18**	23	27	19	21	?	?	?	?	?	...	?	?	?	?

Esta etapa constitui o núcleo do algoritmo mostrado na figura 10.25. O laço externo apenas garante que esta operação será executada para todos os elementos das posições de 1 até N.

O laço interno foi implementado de tal forma que, ao mesmo tempo em que se localiza o lugar certo do elemento da vez, já se abre espaço no vetor. O laço é controlado por dois testes, um deles para garantir que o algoritmo não extrapole o início do vetor, o outro que compara dois elementos e troca de posição sempre que for detectado que o elemento está na posição incorreta.

```
procedure insercao (var v: vetor_r; n: integer);
var i, j: integer;
    aux: real;

begin
     for i:= 1 to n do
     begin
          aux:= v[i];

          (* abre espaco no vetor enquanto localiza a posicao certa *)
          j:= i - 1;
          while (j >= 1) and (v[j] > aux) do
          begin
               v[j+1]:= v[j];
               j:= j - 1;
          end;
          v[j+1]:= aux;
     end;
end;
```

Figura 10.25: Método de ordenação por inserção.

Analisar quantas comparações são feitas é bem mais complicado neste algoritmo, pois isto depende da configuração do vetor de entrada. Neste nosso exemplo, vimos que cada etapa teve um comportamento diferente das outras. Em uma vez o elemento já estava em seu lugar. Em duas outras vezes tivemos que percorrer todo o subvetor inicial, pois os elementos deveriam ser o primeiro, em cada etapa.

Aparentemente, no pior caso possível, que é quando o vetor está na ordem inversa da ordenação, haverá o maior número de comparações, que é quadrático. Mas, na prática, este algoritmo aparenta ser mais rápido que o método da seleção na maior parte dos casos, pois algumas vezes o elemento muda pouco de posição.[4]

10.1.5 Outros algoritmos com vetores

Nesta seção vamos apresentar alguns problemas interessantes que podem ser resolvidos usando-se a estrutura de vetores.

Permutações

Vamos apresentar um problema matemático conhecido como *permutação*, propor uma representação computacional em termos de vetores, e, em seguida, estudar alguns problemas interessantes do ponto de vista de computação.[5]

Os matemáticos definem uma permutação de algum conjunto como uma função bijetora de um conjunto nele mesmo. Em outras palavras, é uma maneira de reordenar os elementos do conjunto. Por exemplo, podemos definir uma permutação do conjunto $\{1,\ 2,\ 3,\ 4,\ 5\ \}$ assim: $P(1) = 4, P(2) = 1, P(3) = 5, P(4) = 2, P(5) = 3$. Esquematicamente temos:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Outra maneira seria: $P(1) = 2, P(2) = 5, P(3) = 1, P(4) = 3, P(5) = 2$. Esquematicamente:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 2 \end{pmatrix}$$

De fato, existem $n!$ maneiras de se reordenar os elementos e obter uma permutação válida. Se $n = 5$ então existem 120 permutações.

Modelando permutações

O primeiro problema interessante do ponto de vista algoritmico é *como representar*

[4] O site `http://cg.scs.carleton.ca/~morin/misc/sortalg` permite visualizar o comportamento dos principais algoritmos de ordenação através de animações. Os dois algoritmos aqui explicados estão lá, entre outros.

[5] Esta seção foi inspirada em uma preparatória para a maratona de programação da ACM da Ural State Universisy (Internal Contest October'2000 Junior Session) encontrada na seguinte URL: `http://acm.timus.ru/problem.aspx?space=1&num=1024`.

uma permutação. Para isto pode-se usar um vetor de n posições inteiras, onde cada posição é um valor (sem repetição) entre 1 e n. Em todos os algoritmos desta seção consideraremos que:

```
const min_i = 1; max_i = 5;
```

Assim os dois vetores que representam as permutações acima são, respectivamente:

1	2	3	4	5
4	1	5	2	3

1	2	3	4	5
2	5	1	3	2

Testando permutações válidas

Uma vez resolvido o problema da representação, podemos estudar o próximo desafio, que é *como testar se um dado vetor (lido do teclado) é uma permutação válida*?

O algoritmo tem que testar se, para os índices de 1 a n, seus elementos são constituídos por todos, e apenas, os elementos entre 1 e n, em qualquer ordem. A função da figura 10.26 apresenta uma possível solução.

```
function testa_permutacao (var v: vetor_i; n: integer): boolean;
var i, j: integer;
    eh_permutacao: boolean;

begin
     eh_permutacao:= true;
     i:= 1;
     while eh_permutacao and (i <= n) do
     begin
          j:= 1;                    (* procura se i esta no vetor *)
          while (v[j] <> i) and (j <= n) do
               j:= j + 1;
          if v[j] <> i then         (* se nao achou nao eh permutacao *)
               eh_permutacao:= false;
          i:= i +1;
     end;
     testa_permutacao:= eh_permutacao;
end; (* testa_permutacao *)
```

Figura 10.26: Verifica se um vetor define uma permutação.

Este algoritmo testa para saber se cada índice entre i e n está presente no vetor. Para isto executa no pior caso algo da ordem do quadrado do tamanho do vetor.

No primeiro semestre de 2011 um estudante[6] sugeriu que basta usar um vetor

[6]Bruno Ricardo Sella

auxiliar, inicialmente zerado, e percorrer o vetor candidato a permutação apenas uma vez. Para cada índice, tentar inserir seu respectivo conteúdo no vetor auxiliar: se estiver com um zero, inserir, senão, não é permutação. Se o vetor auxiliar for totalmente preenchido então temos um vetor que representa uma permutação. Este processo é linear e está ilustrado na figura 10.27.

```
function testa_permutacao_v2 (var v: vetor_i; n: integer): boolean;
var i: integer;
    aux: vetor_i;
    eh_permutacao: boolean;

begin
     zerar_vetor_i (aux,n);
     eh_permutacao:= true;
     i:= 1;
     while eh_permutacao and (i <= n) do
     begin
          if (v[i] >= 1) AND (v[i] <= n) AND (aux[v[i]] = 0) then
               aux[v[i]]:= v[i]
          else
               eh_permutacao:= false;
          i:= i + 1;
     end;
     testa_permutacao_v2:= eh_permutacao;
end; (* testa_permutacao_v2 *)
```

Figura 10.27: Verifica linearmente se um vetor define uma permutação.

Outros estudantes sugeriram uma conjectura, não provada em sala, de que, se todos os elementos pertencem ao intervalo $1 \leq v[i] \leq n$ e $\sum_{i=1}^{n} v[i] = \frac{n(n+1)}{2}$ e ao mesmo tempo $\prod_{i=1}^{n} v[i] = n!$, então o vetor representa uma permutação. Também não encontramos contra-exemplo e o problema ficou em aberto.

Gerando permutações válidas

O próximo problema é gerar aleatoriamente uma permutação. Para isto faremos uso da função *random* da linguagem *Pascal*.

O primeiro algoritmo gera um vetor de maneira aleatória e depois testa se o vetor produzido pode ser uma permutação usando o código da função *testa_permutacao* já implementado. A tentativa é reaproveitar código a qualquer custo. Este raciocínio está implementado no código da figura 10.28.

Este algoritmo é absurdamente lento quando n cresce. Isto ocorre pois os vetores são gerados e depois testados para ver se são válidos, mas, conforme esperado, é muito provável que números repetidos sejam gerados em um vetor com grande número de elementos. Um dos autores deste material teve paciência de esperar o código terminar apenas até valores de n próximos de 14. Para valores maiores o código ficou infernalmente demorado, levando várias horas de processamento.

Numa tentativa de melhorar o desempenho, o que implica em abrir mão da como-

```
procedure gerar_permutacao (var v: vetor_i; n: integer);
var i: integer;

begin
      randomize;
      repeat     (* repete ate conseguir construir uma permutacao valida *)
            for i:= 1 to n do
                  v[i]:= random (n) + 1;   (* sorteia numero entre 1 e n *)
      until testa_permutacao_v2 (v, n);
end; (* gera_permutacao *)
```

Figura 10.28: Gerando uma permutação, versão 1.

didade de se aproveitar funções já existentes, este mesmo autor pensou que poderia gerar o vetor de modo diferente: gerar um a um os elementos e testar se eles já não pertencem ao conjunto já gerado até a iteração anterior. Isto garante que o vetor final produzido é válido. A procedure da figura 10.29 apresenta a implementação desta nova ideia.

```
procedure gerar_permutacao_v2 (var v: vetor_i; n: integer);
var i, j: integer;

begin
      randomize;
      v[1]:= random (n) + 1;
      for i:= 2 to n do
            repeat
                  v[i]:= random (n) + 1;   (* gera um numero entre 1 e n *)
                  j:= 1; (* procura se o elemento ja existe no vetor *)
                  while (j < i) and (v[i] <> v[j]) do
                        j:= j + 1;
            until j = i;   (* descobre que o elemento eh novo *)
end; (* gera_permutacao_v2 *)
```

Figura 10.29: Gerando uma permutação, versão 2.

Este algoritmo executa na casa de 2 segundos para vetores de tamanho próximos de 1000, mas demora cerca de 30 segundos para entradas de tamanho que beiram os 30.000. Para se pensar em tamanhos maiores a chance do tempo ficar insuportável é enorme. Mas já é melhor do que o anterior.

Queremos gerar um vetor que represente uma permutação, provavelmente para fins de testes. Uma maneira possível seria a seguinte: inicializa-se um vetor de forma ordenada, depois faz-se alterações aleatórias de seus elementos um número também aleatório de vezes. Esta ideia foi implementada e é mostrada na figura 10.30.

Este código produz corretamente vetores que representam permutações com bom grau de mistura dos números em tempo praticamente constante para entradas da ordem de um milhão de elementos (usando-se o tipo *longint*).[7]

[7]Se alguém souber de um modo mais eficiente de gerar uma permutação, favor avisar. Não só

```
procedure gerar_permutacao_v3 (var v: vetor_i; n: integer);
var i, j, k, aux, max: integer;
begin
      for i:= 1 to n do
            v[i] := i;

      randomize;
      max:= random (1000); (* vai trocar dois elementos de 0 a 999 vezes *)
      for i:= 1 to max do
      begin
            j:= random (n) + 1;
            k:= random (n) + 1;
            aux:= v[j];
            v[j]:= v[k];
            v[k]:= aux;
      end;
end; (* gera_permutacao_v3 *)
```

Figura 10.30: Gerando uma permutação, versão 3.

Em uma aula do segundo semestre de 2010 surgiu uma nova ideia para se gerar um vetor de permutação.[8]

A sugestão é modificar o algoritmo 10.29, fazendo com que um vetor auxiliar contenha os números ainda não colocados no vetor permutação. O sorteio deixa de ser sobre o elemento a ser inserido, mas agora sobre o índice do vetor auxiliar, cujo tamanho decresce na medida em que os números vão sendo sorteados. O código da figura 10.31 ilustra estas ideias. Ele foi implementado durante a aula e possibilitou gerar vetores de tamanhos incrivelmente grandes em tempo extremamente curto.[9]

Determinando a ordem de uma permutação

Antes de apresentarmos o próximo problema do ponto de vista algoritmico a ser tratado precisamos introduzi-lo do ponto de vista matemático.

Observem que, uma vez que a função que define a permutação é sobre o próprio conjunto, ocorre que, se $P(n)$ é uma permutação, então $P(P(n))$ também é. Logo, é possível calcular o valor de expressões tais como $P(P(1))$. De fato, consideremos novamente a permutação:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

Então pode-se calcular:

- $P(P(1)) = P(4) = 2$.

daremos os créditos necessários como também mostraremos os resultados aqui neste material.

[8] Créditos para o Felipe Z. do Nascimento.

[9] Código e testes feitos na aula por Renan Vedovato Traba, a partir da ideia do Felipe.

```
procedure gerar_permutacao_v4 (var v: vetor_i; n: longint);
var i, j, tam: longint;
    aux: vetor_i;

begin
     for i := 1 to n do
          aux[i] := i;

     randomize;
     tam:= n;
     for i := 1 to n do
     begin
          j := random(tam) + 1;
          v[i] := aux[j];
          aux[j] := aux[tam];
          tam := tam - 1;
      end;
end; (* gera_permutacao_v4 *)
```

Figura 10.31: Gerando uma permutação, versão 4.

- $P(P(2)) = P(1) = 4$.
- $P(P(3)) = P(5) = 3$.
- $P(P(4)) = P(2) = 1$.
- $P(P(5)) = P(3) = 5$.

Desta maneira, definimos $P^2(n) = P(P(n))$. Em termos gerais, podemos definir o seguinte:

$$\begin{cases} P^1(n) = P(n); \\ P^k(n) = P(P^{k-1}(n)) \quad k \geq 2. \end{cases}$$

Dentre todas as permutações, existe uma especial:

$$ID = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Isto é, $P(i) = i, \forall i$. Esta permutação recebe um nome especial ID. É possível demonstrar que, para quaisquer k e n, $ID^k(n) = ID(n)$. Também é possível demonstrar que a sentença seguinte também é válida:

Seja $P(n)$ uma permutação sobre um conjunto de n elementos. Então existe um número natural k tal que $P^k = ID$. Este número natural é chamado da *ordem* da permutação.

Vamos considerar como exemplo a permutação acima. Podemos calcular para valores pequenos de k como é P^k:

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

$$P^2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 3 & 1 & 5 \end{pmatrix}$$

$$P^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 4 & 3 \end{pmatrix}$$

$$P^4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 3 & 2 & 5 \end{pmatrix}$$

$$P^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

$$P^6 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Isto é, a ordem da permutação P é 6.

Chegamos no ponto de apresentarmos o próximo problema[10]. Dada uma permutação, encontrar sua ordem. Simular a sequência de operações e testar quando a identidade for encontrada, contando quantas operações foram feitas é muito caro. Tem que haver uma maneira melhor.

A função da figura 10.32 implementa um algoritmo que recebe como entrada uma permutação (válida) e retorna sua ordem.

Este algoritmo parte da ideia de que cada elemento $P(i) = x$ do conjunto retorna à posição i ciclicamente, de *cont* em *cont* permutações. Ou seja, $P^{cont}(i) = x, P^{2\times cont}(i) = x, \ldots$. O mesmo ocorre para todos elementos do conjunto, mas cada um possui um ciclo (valor de *cont*) próprio.

```
function ordem_permutacao (var v: vetor_i; n: integer): int64;
var  mmc, cont: int64;
     p, i: integer;
begin
     mmc := 1;
     for i := 1 to n do
     begin
          cont := 1;
          p := i;
          while (v[p] <> i) do
          begin
               cont:= cont + 1;
               p := v[p];
          end;
          mmc := mmc * cont div mdc(mmc, cont);
     end;
     ordem_permutacao:= mmc;
end;
```

Figura 10.32: Calcula a ordem de uma permutação.

[10]Este é o problema da Maratona da ACM.

Para exemplificar, tomemos a permutação acima. Para o índice 1, temos que $P^3(1) = 1$. Isto quer dizer que para todo múltiplo de 3 (a cada 3 iterações) é verdade que $P^{3k}(1) = 1$. Isto também ocorre para os índices 2 e 4. Mas para os índices 3 e 5, o número de iterações para que ocorra uma repetição é de duas iterações. Logo, pode-se concluir que a permutação ID ocorrerá exatamente na iteração que é o mínimo múltiplo comum (MMC) entre o número que provoca repetição entre todos os índices. Observamos que:

$$MMC(x_1, x_2, \ldots, x_n) = MMC(x_1, MMC(x_2, \ldots, x_n).$$

Infelizmente, não existe algoritmo eficiente para cálculo do MMC. Mas existe para o cálculo do MDC (máximo divisor comum). De fato, implementamos o algoritmo de Euclides (figura 6.16, seção 6.4) e mostramos que ele é muito eficiente. Felizmente, a seguinte propriedade é verdadeira:

$$MDC(a, b) = \frac{a \times b}{MMC(a, b)}$$

O programa acima explora este fato e torna o código muito eficiente para calcular a ordem de permutações para grandes valores de n. O estudante é encorajado aqui a gerar uma permutação com os algoritmos estudados nesta seção e rodar o programa para valores de n compatíveis com os tipos de dados definidos (integer).

Polinômios

Nesta seção vamos mostrar como representar e fazer cálculos com polinômios representados como vetores.

Para uma sucessão de termos $a_0, ..., a_n \in \mathbb{R}$, podemos para este curso definir um polinômio de grau n como sendo uma função que possui a seguinte forma:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Vamos considerar ao longo desta seção que $\forall k > n$, então $a_k = 0$. Também consideramos que $a_n \neq 0$ para um polinômio de grau n.

Do ponto de vista computacional, uma possível representação para um polinômio é um vetor de $n+1$ elementos cujos conteúdos são os coeficientes reais dos respectivos monômios. Consideremos então o tipo abaixo, podemos exemplificar alguns casos:

```
type polinomio = array [0..max] of real;
```

- $P(x) = 5 - 2x + x^2$:

0	1	2
5	-2	1

- $P(x) = 7 - 2x^2 + 8x^3 - 2x^7$

0	1	2	3	4	5	6	7
7	0	-2	8	0	0	0	-2

No restante desta seção iremos mostrar algoritmos que realizam operações costumeiras sobre polinômios. A primeira é calcular o valor de um polinômio em um dado ponto $x \in \mathbb{R}$. Por exemplo, se $P(x) = 5 - 2x + x^2$ então, $P(1) = 5 - 2 \times 1 + 1^2 = 4$. A função em *Pascal* que implementa este cálculo está apresentado na figura 10.33.

```
function valor_no_ponto (var p: polinomio; graup: integer; x: real): real;
var soma, potx: real;
    i: integer;
begin
  potx:= 1;
  soma:= 0;
  for i:= 0 to graup do
  begin
    soma:= soma + p[i]*potx;
    potx:= potx * x;
  end;
  valor_no_ponto:= soma;
end;
```

Figura 10.33: Calcula o valor de $P(x)$ para um dado $x \in \mathbb{R}$.

O próximo algoritmo interessante é o cálculo do polinômio derivada de um polinômio P. Seja $P\prime(x)$ a derivada de $P(x)$ assim definido:

$$P\prime(x) = na_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \ldots + 2a_2 x + a_1$$

O programa que implementa este cálculo está na figura 10.34.

```
procedure derivar (var p: polinomio;      graup: integer;
                   var d: polinomio; var graud: integer);
var i: integer;
begin
  if graup = 0 then
  begin
    graud:= 0;
    d[0]:= 0;
  end
  else
  begin
    graud:= graup - 1;
    for i:= 0 to graud do
      d[i]:= (i+1) * p[i+1];
  end;
end;
```

Figura 10.34: Calcula o polinômio derivada de $P(x)$.

Por outro lado, para calcular o valor no ponto de uma derivada de um polinômio, não é necessário que se calcule previamente um vetor auxiliar contendo a derivada. Isto pode ser feito diretamente usando-se o polinômio P, bastando trabalhar corretamente os índices do vetor, conforme mostrado na figura 10.35.

```
function valor_derivada_no_ponto (var p: polinomio; graup: integer; x: real): real;
var i: integer;
    soma, potx: real;
begin
     soma:= 0;
     potx:= 1;
     for i:= 1 to graup do
     begin
          soma:= soma + i * p[i] * potx;
          potx:= potx * x;
     end;
     valor_derivada_no_ponto:= soma;
end;
```

Figura 10.35: Calcula o valor de $P\prime(x)$ para um dado $x \in \mathbb{R}$.

Os próximos problemas vão nos permitir trabalhar um pouco com os índices do vetor. O primeiro problema é a soma de polinômios. O segundo é a multiplicação. Antes mostraremos a definição matemática para estes conceitos.

Sejam dois polinômios P e Q assim definidos, supondo que $n >= m$:

$$P(x) = a_n x^n + \ldots + a_m x^m + \ldots + a_1 x + a_0$$

$$Q(x) = b_n x^m + b_{n-1} x^{m-1} + \ldots + b_1 x + b_0$$

Então o polinômio soma de P e Q, denotado $P + Q$ é assim definido:

$$(P+Q)(x) = a_n x^n + \ldots + a_{m+1} x^{m+1} + (a_m + b_m) x^m + \ldots + (a_1 + b_1) x + (a_0 + b_0)$$

Basicamente é a mesma operação de soma de vetores estudada neste capítulo, embora naquele caso exigimos que os tamanhos dos vetores fossem iguais. No caso de polinômios os vetores podem ter tamanhos diferentes desde que se assuma que os coeficientes que faltam no polinômio de maior graus são nulos. A implementação deste processo está na figura 10.36.

Considerando os mesmos polinômios P e Q acima definidos, o produto de P por Q, denotado PQ é assim definida:

$$\begin{aligned}(PQ)(x) = (a_{n+m} b_0 + \ldots + a_n b_m + \ldots + a_0 b_{n+m}) x^{n+m} + \ldots + \\ (a_k b_0 + a_{k-1} b_1 + \ldots + a_0 b_k) x^k + \ldots + (a_1 b_0 + a_0 b_1) x + (a_0 b_0)\end{aligned}$$

```
procedure somar (var p: polinomio;      graup: integer;
                 var q: polinomio;      grauq: integer;
                 var s: polinomio; var graus: integer);
var i,menorgrau: integer;
begin
  (* o grau do pol soma eh o maior grau entre p e q *)
  (* copiar os coeficientes que o maior pol tem a mais *)
  if graup > grauq then
  begin
    graus:= graup;
    menorgrau:= grauq;
    for i:= menorgrau+1 to graus do
      s[i]:= p[i];
  end
  else
  begin
    graus:= grauq;
    menorgrau:= graup;
    for i:= menorgrau+1 to graus do
      s[i]:= q[i];
  end;

  for i:= 0 to menorgrau do
    s[i]:= p[i] + q[i];

end;
```

Figura 10.36: Calcula a soma de $P(x)$ com $Q(x)$.

A operação matemática exige que sejam feitas todas as multiplicações e posterior agrupamento dos monômios de mesmo grau, somando-se os coeficientes, para cada monômio.

O programa apresentado na figura 10.37 implementa os cálculos para obtenção do polinômio produto de P por Q. O programa realiza os cálculos para cada monômio à medida em que os índices dos dois comandos *for* variam, o que é um uso especial da técnica dos acumuladores, embora os acúmulos não sejam simultâneos para cada monômio do resultado final da operação, eles são feitos aos poucos. Para isto é preciso zerar o vetor antes de começar.

```
procedure multiplicar (var p: polinomio;      graup: integer;
                       var q: polinomio;      grauq: integer;
                       var m: polinomio; var grauM: integer);
var i,j: integer;
begin
  grauM:= graup + grauq;
  for i:= 0 to grauM do
    m[i]:= 0;

  for i:= 0 to graup do
    for j:= 0 to grauq do
      m[i+j]:= m[i+j] + p[i]*q[j];

  if ((graup = 0) and (p[0] = 0)) or
     ((grauq = 0) and (q[0] = 0)) then
    grauM:= 0;
end;
```

Figura 10.37: Calcula o produto de $P(x)$ com $Q(x)$.

10.1.6 Exercícios

1. Faça um programa que leia e armazene em um vetor uma sequência de inteiros. Em seguida o programa deve ler uma sequência de inteiros informados pelo usuário e, para cada um deles, dizer se ele pertence ou não ao vetor armazenado previamente.

2. Faça um programa que leia duas sequências de n inteiros em dois vetores distintos, digamos, v e w e verifique se os dois vetores são idênticos.

3. Faça um programa em que leia dois vetores de números reais e descubra se um deles é permutação do outro, isto é, se eles tem os mesmos elementos, ainda que em ordem diferente. A quantidade de elementos lidos em cada vetor é no máximo 100, e cada sequência termina quando o valor 0 é digitado. Por exemplo:

   $[2, 2, 0, 3, 4]$ e $[2, 2, 0, 3, 4]$: sim.

   $[2, 2, 0, 3, 4]$ e $[4, 3, 2, 0, 2]$: sim.

   $[2, 2, 0, 3, 4]$ e $[4, 3, 4, 0, 2]$: não.

   $[3, 0, 5]$ e $[3, 0, 5, 3]$: não.

   Implemente três versões deste problema:

   - ordenando os vetores para em seguida compará-los;
   - sem ordenar os vetores;
   - crie uma função que retorna 0 se x não pertence a v e caso contrário retorna o índice do vetor onde x se encontra. Use esta função para resolver este problema.

4. Faça um programa que leia duas sequências de inteiros, não necessariamente contendo a mesma quantidade de números. Seu programa deverá:

   - dizer se a segunda sequência está contida na primeira. Exemplo:

     ```
     v1: 7 3 2 3 2 6 4 7
     v2: 3 2 6
     Saída: sim
     ```

   - constrir um terceiro vetor, sem destruir os originais, que é a concatenação do primeiro com o segundo;

     ```
     v1: 7 3 2 6
     v2: 5 1 8 4 9
     Saída: 1 2 3 4 5 6 7 8 9
     ```

   - ordená-los, e em seguida imprimir todos os números ordenados em ordem crescente. Exemplo:

     ```
     v1: 7 3 2 6
     v2: 5 1 8 4 9
     Saída: 1 2 3 4 5 6 7 8 9
     ```

5. Crie uma função em que receba um vetor de inteiros de tamanho n e devolva o valor *true* se o vetor estiver ordenado e *false* em caso contrário.

6. Aproveitando as soluções dos problemas anteriores, escreva um programa em que leia dois vetores de inteiros v e w, de dimensões m e n respectivamente, verifique se eles estão ordenados, ordene-os em caso contrário e, em seguida, imprima a intercalação dos dois.
   Exemplo de intercalação: v: 1 4 6 9; w: 2, 3, 5, 7.
   Saída: 1, 2, 3, 4, 5, 6, 7, 9.

7. Dados dois números naturais m e n, uma frase com m letras e uma palavra com n letras, escreva um procedimento que determine o número de vezes que a palavra ocorre na frase e a posição em que cada ocorrência inicia.

   Exemplo:

   ```
   Para M = 30, N = 3, a palavra ANA e a frase:
       ANA E MARIANA GOSTAM DE BANANA
   A palavra ANA ocorre 4 vezes, nas posições 1, 11, 26, 28.
   ```

8. Dada uma sequência de N números, determinar quantos números dintintos compõe a sequência e o número de vezes que cada um deles ocorre na mesma. Exemplo:

   N=5 1 2 3 2 3 a sequência tem três números distintos, 1, 2 e 3. Ocorrências: 1 ocorre 1 vez 2 ocorre 2 vezes 3 ocorre 2 vezes

9. Dadas duas sequências com n números inteiros entre 0 e 1, interpretados como números binários:

   (a) imprimir o valor decimal dos números;

   (b) calcular a soma de ambos (em binário), usando o "vai-um";

   (c) imprimir o valor decimal da soma.

10. Escreva um programa em que leia os seguintes valores: um inteiro B, um inteiro N ($1 \leq N \leq 10$), e N valores inteiros. A ideia é que estes valores sejam entendidos como a representação de um número não negativo na base B. Estes valores deverão ser inseridos em um vetor de tamanho $N + 1$, onde a primeira posição armazena a base B e as outras N posições o restante dos números lidos. Note que o intervalo de valores possíveis para cada dígito na base B é $[0, B-1]$. Seu programa deve retornar o valor em decimal do número representado no vetor. Se o número representado no vetor não for válido na base B então deverá ser retornado o código de erro "-1". Por exemplo, se $B = 3$ o número 2102 na base 3 equivale ao valor decimal 65; se $B = 4$ o número 35 é inválido na base 4.

11. Faça um programa em que, dadas duas sequências com N números inteiros entre 0 e 9, interpretadas como dois números inteiros de N algarismos, calcular a sequência de números que representa a soma dos dois inteiros, usando o "vai-um". Por exemplo:

```
N=6,
        4 3 4 2 5 1
      + 7 5 2 3 3 7
      1 1 8 6 5 8 8
```

12. Dada uma sequência x_1, x_2, ..., x_n de números inteiros, determinar um segmento de soma máxima. Exemplo: na sequência 5, 2, -2, -7, 3, 14, 10, -3, 9, -6, 4, 1, a soma do maior segmento é 33, obtida pela soma dos números de 3 até 9.

13. Implemente um programa que leia um vetor de 1 milhão de inteiros em um vetor de inteiros. Gere um número aleatório e o procure neste vetor de duas maneiras diferentes: uma usando busca com sentinela e outra usando busca binária. Seu programa deve imprimir uma tabela com o número de comparações feitas em cada um dos casos acima (busca com sentinela e busca binária). Desconsidere o tempo gasto com ordenação no caso da busca binária. A busca com sentinela deve ser feita em um vetor não ordenado. Gere 200 números aleatórios e imprima a média de comparações para cada um dos dois algoritmos sendo testados.

14. Suponha que um exército tenha 20 regimentos e que eles estão em processo de formação. Inicialmente o primeiro tem 1000 homens, o segundo 950, o terceiro 900, e assim por diante, até o vigésimo que tem 50. Suponhamos que a cada semana 100 homens são enviados para cada regimento, e no final da semana o maior regimento é enviado para o *front*. Imaginemos que o general do quinto regimento é companheiro de xadrez do comandante supremo, e que eles estão no meio de uma partida. O comandante supremo então envia apenas 30 homens para o quinto regimento a cada semana, esperando com isto poder acabar o jogo com seu colega. Escreva um programa em que diga, a cada semana, qual é o regimento enviado ao *front* e mostre o *status* dos outros regimentos. O programa deve também determinar exatamente quantas semanas levará o quinto regimento para ser deslocado ao *front*.

15. Suponha que você esteja usando o método da ordenação por seleção. Qual das sequências abaixo requerirá o menor número de trocas? Quantas? Qual requerirá o maior número de trocas? Quantas? Explique.

    (a) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

    (b) 5, 4, 3, 2, 1, 10, 9, 8, 7, 6.

    (c) 10, 1, 9, 2, 8, 3, 7, 4, 6, 5.

    (d) 2, 3, 4, 5, 6, 7, 8, 9, 10, 1.

    (e) 1, 10, 2, 9, 3, 8, 4, 7, 5, 6.

16. Suponha que você tem uma variável do tipo vetor declarada como: *array [1..50] of real;*. Faça uma função que inicialize o vetor de modo que os elementos de índices ímpares recebam o valor inicial -2.0 e os elementos de índices pares recebam o valor inicial 7.0. Sua função deve fazer uso de apenas um comando de repetição, que incrementa de um em um, e de nenhum comando de desvio condicional.

17. Qual dos seguintes problemas requer o uso de vetores para uma solução elegante?

    (a) Ler cerca de duzentos números e imprimir os que estão em uma certa faixa;

    (b) Computar a soma de uma sequência de números;

    (c) Ler exatamente duzentos números e ordená-los em ordem crescente;

    (d) Encontrar o segundo menor elemento de uma sequência de entrada;

    (e) Encontrar o menor inteiro de uma sequência de inteiros.

18. Considere um vetor declarado como: *array [1..50] of integer* que tem a particularidade de todos os elementos estarem entre 1 e 30, sendo que nenhum é repetido. Faça um programa que ordene o vetor de maneira eficiente explorando esta característica e fazendo o menor número possível de trocas.

19. Dada uma sequência $x_1, x_2, \ldots, x_k$ de números reais, verifique se existem dois segmentos consecutivos iguais nesta sequência, isto é, se existem i e m tais que:

$$x_i, x_{i+1}, \ldots, x_{i+m-1} = x_{i+m}, x_{i+m+1}, \ldots, x_{i+2m-1}.$$

    Imprima, caso existam, os valores de i e de m. Caso contrário, não imprima nada. Exemplo: Na sequência 7,9,5,4,5,4,8,6, existem $i = 3$ e $m = 2$.

20. Um coeficiente binomial, geralmente denotado $\binom{n}{k}$, representa o número de possíveis combinações de n elementos tomados k a k. Um "Triângulo de Pascal", uma homenagem ao grande matemático Blaise Pascal, é uma tabela de valores de coeficientes combinatoriais para pequenos valores de n e k. Os números que não são mostrados na tabela têm valor zero. Este triângulo pode ser construído automaticamente usando-se uma propriedade conhecida dos coeficientes binomiais, denominada "fórmula da adição": $\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}$. Ou seja, cada elemento do triângulo é a soma de dois elementos da linha anterior, um da mesma coluna e um da coluna anterior. Veja um exemplo de um triângulo de Pascal com 7 linhas:

$$\begin{array}{ccccccc} 1 & & & & & & \\ 1 & 1 & & & & & \\ 1 & 2 & 1 & & & & \\ 1 & 3 & 3 & 1 & & & \\ 1 & 4 & 6 & 4 & 1 & & \\ 1 & 5 & 10 & 10 & 5 & 1 & \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{array}$$

Faça um programa em que imprima na tela um triângulo de Pascal com 10 linhas. Seu programa deve obrigatoriamente fazer uso de exatamente dois vetores durante o processo de construção. Um deles conterá a última linha ímpar gerada, enquanto que o outro conterá a última linha par gerada. Lembre-se que os elementos que não aparecem na tabela tem valor nulo. Você deve sempre ter o controle do tamanho da última linha impressa (o tamanho útil dos vetores em cada passo). Você deve também usar um procedimento para imprimir o vetor. Observe que não há entrada de dados, os dois vetores são gerados, um a partir do outro. O único elemento da primeira linha tem o valor 1. Você deve obrigatoriamente declarar um tipo vetor com tamanho máximo Tam_max_vetor, e o seu programa deverá tomar cuidado para manipular corretamente vetores de tamanho menor do que o tamanho máximo, impedindo que haja uma atribuição em posição ilegal de memória.

21. Resolva o problema do triângulo de Pascal usando apenas um vetor.

22. Seja um polinômio $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ de grau $n \geq 2$. Uma possível maneira de calcular uma raiz do polinômio é pelo "método de Newton". Este método consiste em se fornecer uma aproximação inicial para a raiz, isto é, um valor que não é a raiz exata, mas é um valor próximo. Assim, se x_0 é esta aproximação inicial, $p(x_0)$ não é zero mas espera-se que seja próximo de zero. A obtenção da raiz pelo método de Newton é feita pelo refinamento desta solução inicial, isto é, pela tentativa de minimizar o erro cometido. Isto é feito pela expressão seguinte:

$$x_{n+1} = x_n - \frac{p(x_n)}{p\prime(x_n)},$$

$n = 0, 1, 2, \ldots$, e onde $p\prime(x)$ é a primeira derivada de $p(x)$. Usualmente, repete-se este refinamento até que $|x_{n+1} - x_n| < \epsilon, \epsilon > 0$, ou até que m iterações tenham sido executadas.
Construa um programa em que receba como dados de entrada um polinômio $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ e uma aproximação inicial x_0 da raiz de $p(x)$, $\epsilon > 0$ e o número máximo de iterações, e calcule uma aproximação da raiz de $p(x)$ pelo método de Newton. Utilize obrigatoriamente um procedimento que receba como parâmetro um polinômio $p(x)$ (incluindo a informação sobre o grau do polinômio) e que calcule e retorne a função derivada $p\prime(x)$. Utilize também uma função que receba como parâmetros um polinômio $p(x)$ e um valor real $\overline{x}$ e retorne o valor do polinômio no ponto $\overline{x}$, isto é $p(\overline{x})$. Use esta função para calcular, a cada iteração do método de Newton, os valores de $p(x_n)$ e de $p\prime(x_n)$.

23. Faça um programa em que leia uma sequência de 10 letras (caracteres de A a Z), as armazene em um vetor de 10 posições e imprima a lista de letras repetidas no vetor. Sendo assim, para os dados: A J G A D F G A A, a saída deve ser: A G.

24. Escreva o programa da busca binária de um valor x num vetor de inteiros que, ao invés de achar a primeira ocorrência do valor na lista, identifique e imprima

o menor índice do vetor no qual o valor ocorra.

25. Escreva um programa em que leia uma sequência de *código de operação* e *valor*, onde o *código de operação* é um inteiro com os seguintes valores:

    - 0 (zero): fim
    - 1 (um): inserção
    - 2 (dois): remoção

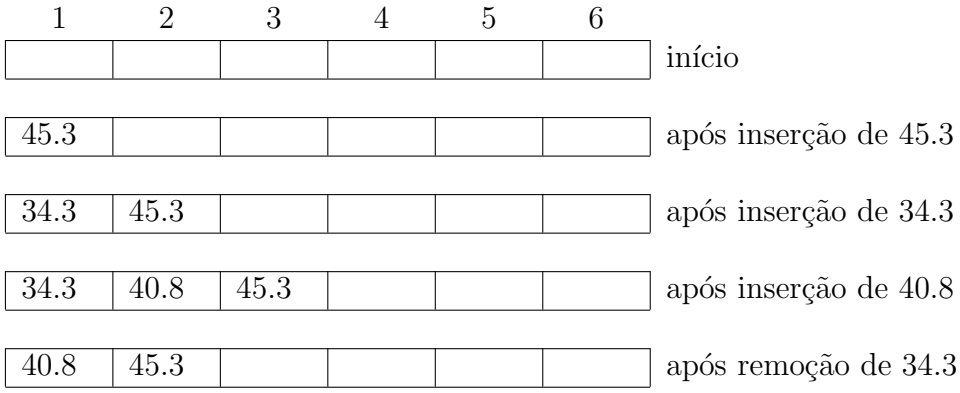
    O *valor* lido é um real que deve ser inserido em um vetor (caso a operação seja 1), ou removido do vetor (caso a operação seja 2). As inserções no vetor devem ser realizadas de forma que o vetor esteja sempre ordenado. No final do programa o vetor resultante deve ser impresso.

    Detalhamento:

    - a quantidade máxima de valores que pode ser inserida é 100;
    - se a quantidade máxima for ultrapassada o programa deve dar uma mensagem de erro;
    - se for requisitada a remoção de um número não existente o programa deve dar uma mensagem de erro;
    - se o código de operação for inválido o programa deve continuar lendo um novo código até que ele seja 0 (zero), 1 (um) ou 2 (dois).

    Exemplo de execução:

```
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 45.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 34.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 40.8
Entre com operacao (0=fim, 1=insercao, 2=remocao): 2
Valor: 34.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 0
Vetor resultante
40.8   45.3
```



26. Escreva um programa que leia duas sequências de caracteres e verifica se a segunda sequência é subpalavra da primeira. Por exemplo, *todo* é subpalavra de *metodo* e *ar* é subpalavra de *farmacia*. Porém, *todo* não é subpalavra de *todavia*. A leitura das sequências deve ser feita caracter por caracter e o final de cada sequência é sinalizada pelo caracter '.'. Se a segunda sequência é uma subpalavra, a saída do programa deve ser a posição na qual ela começa. Caso contrário, escrever a mensagem "Nao eh subpalavra.". Observações:

    - cada sequência tem no máximo 80 caracteres.
    - você não pode utilizar funções de manipulação de cadeias de caracteres existentes no compilador, mas somente as funções para o tipo `char`.

    Exemplo de execução:

    ```
    Entre com duas palavras terminadas por ponto:
    metodo.todo.

    A segunda subpalavra comeca na posicao 3 da primeira.
    ```

27. Escreva um programa em que leia uma sequência de n valores reais ($n \leq 100$) e os insira num vetor. A sequência termina quando o valor lido for 0. O programa deve escrever o valor da divisão da soma dos valores positivos pela soma dos valores negativos que estão armazenados no vetor. Cuidado com divisões por zero.

28. Escreva uma função em que substitui em um texto a primeira ocorrência de uma palavra por outra. A função deve retornar `true` se a substituição for bem sucedida e `false` caso a palavra não seja encontrada no texto. O texto e as palavras são representados por vetores do tipo `char`. Por exemplo:

    ```
            +---+---+---+---+---+---+---+---+---+---+
    texto1  | e | x | e | m | p | r | o |   | u | n |
            +---+---+---+---+---+---+---+---+---+---+
              +---+---+---+---+---+
    palavra1  | r | o |   | u | n |
              +---+---+---+---+---+
              +---+---+---+---+---+---+---+
    palavra2  | l | o |   | d | o | i | s |
              +---+---+---+---+---+---+---+
            +---+---+---+---+---+---+---+---+---+---+---+---+
    texto2  | e | x | e | m | p | l | o |   | d | o | i | s |
            +---+---+---+---+---+---+---+---+---+---+---+---+
    ```

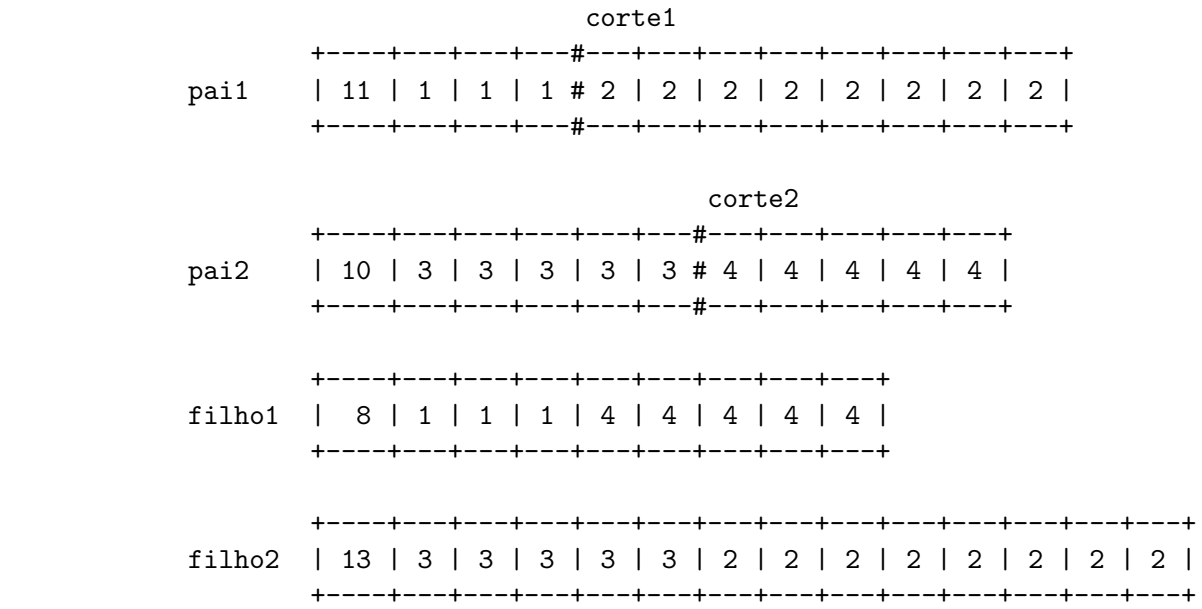
    A função recebe como parâmetros o texto, a palavra a ser substituída e a nova palavra. No exemplo, `texto1` mostra o estado inicial do texto e `texto2` o estado do texto após a substituição da `palavra1` pela `palavra2`.

    Você pode usar, caso seja necessário, a função:

    ```
    buscachar(texto, pos, letra);
    ```

que busca um caractere (`letra`) a partir de uma determinada posição (`pos`) em um vetor que contém o texto (`texto`). A função `buscaletra` retorna a posição no vetor `texto` da primeira ocorrência de `letra`, se `letra` não aparece no texto a função retorna `-1`.

29. Um *algoritmo genético* é um procedimento computacional de busca, inspirado no processo biológico de evolução, que otimiza a solução de um problema. O problema é modelado por: uma população de indivíduos que representam possíveis soluções; uma função que avalia a qualidade da solução representada por cada indivíduo da população e um conjunto de operadores genéticos. Os indivíduos são dados por sequências de genes que representam características da solução do problema. O procedimento consiste em aplicar os operadores genéticos sobre a população, gerando novos indivíduos e selecionar os mais aptos para constituirem uma nova população. Esse processo é repetido até que uma solução adequada seja obtida. Dentre os operadores genéticos, o mais importante é o de recombinação genética (*crossover*) de dois indivíduos. Esse operador corta em duas partes as sequências de genes de dois indivíduos pais (`pai1` e `pai2`) e gera dois novos indivíduos filhos (`filho1` e `filho2`). `filho1` é dado pela contatenação da primeira parte dos genes de `pai1` com a segunda parte de `pai2` e `filho2` pela concatenação da primeira parte de `pai2` com a segunda parte de `pai1`. O diagrama abaixo exemplifica a operação em indivíduos representados por vetores de números inteiros onde a primeira posição contém o tamanho do vetor:



Escreva um procedimento em que execute a operação de recombinação descrita acima, usando a estrutura de dados vetor. O procedimento deve receber seis parâmetros, um vetor representando o primeiro pai, a posição de corte no primeiro pai, um vetor representando o segundo pai, a posição do corte no segundo pai, e dois vetores que receberão os novos indivíduos. No exemplo apresentado a chamada do procedimento seria:

```
corte1 := 4;
corte2 := 6;
crossover(pai1, corte1, pai2, corte2, filho1, filho2);
```

Note que os vetores devem iniciar na posição zero e essa posição é usada para armazenar o tamanho do vetor. No caso do exemplo, `pai1[0]=11`, `pai2[0]=10`, `filho1[0]=8` e `filho2[0]=13`. Os pontos de corte devem estar dentro dos vetores: `1 < corte1 <= pai1[0]` e `1 < corte2 <= pai2[0]`.

30. Escreva um procedimento em que implemente a subtração de números binários. Considere que os números binários têm N bits e que os bits são armazenados em vetores de inteiros de N posições indexadas de 1 a N. O primeiro bit do vetor representa o sinal do número, sendo zero (0) para o números positivos e um (1) para negativos. Os demais bits representam o valor absoluto do número. Por exemplo, para $N = 11$, os números decimais -13, 12 e 1010 são representados pelos seguintes vetores:

```
      +---+---+---+---+---+---+---+---+---+---+---+
-13:  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
      +---+---+---+---+---+---+---+---+---+---+---+

      +---+---+---+---+---+---+---+---+---+---+---+
 12:  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
      +---+---+---+---+---+---+---+---+---+---+---+

      +---+---+---+---+---+---+---+---+---+---+---+
1010:  | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
      +---+---+---+---+---+---+---+---+---+---+---+
```

O procedimento recebe dois vetores do mesmo tamanho como parâmetros e deve gerar como resultado um vetor que contenha a subtração do primeiro pelo segundo. Por exemplo $-12 - 1010 = -1022$:

```
      +---+---+---+---+---+---+---+---+---+---+---+
-998:  | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
      +---+---+---+---+---+---+---+---+---+---+---+
```

Caso o resultado tenha mais bits que o espaço disponível no vetor o procedimento deve descartar os bits excedentes.

31. Escreva um programa em que leia uma sequência de N valores reais não nulos ($N \leq 100$) e os insira em um vetor. A sequência termina quando o valor lido for 0. O elemento zero não faz parte do vetor. Leia do teclado um inteiro p ($p \leq N$) e considere que o elemento $V[p]$ como o pivô na operação de rearranjar o vetor de tal maneira que todos os elementos à esquerda de $V[p]$ sejam menores que ele e todos os da direta sejam maiores ou iguais a ele. Por exemplo, considere o seguinte vetor dado como entrada:

```
+------+------+------+------+------+------+------+
| 99.7 | 32.6 | 2.45 | 13.4 | 26.7 | 12.2 | 0.51 |
+------+------+------+------+------+------+------+
```

e P valendo 3, o programa deve gerar como resultado um vetor onde todos os elementos que estão à esquerda do valor 2.45 no vetor sâo menores que ele, enquanto que os da direita são maiores do que ele.

32. Escreva uma função em que procura uma palavra dentro de um texto. A função deve receber como parâmetros:

    - um vetor do tipo `texto` que contém um texto;
    - o tamanho do vetor que contém o texto;
    - a posição inicial da busca dentro do vetor que contém o texto;
    - um vetor do tipo `texto` que contém uma palavra;
    - o tamanho do vetor que contém a palavra.

    A função deve retornar um número inteiro indicando a posição no texto onde a palavra foi encontrada pela primeira vez.

    Caso a palavra não seja encontrada ou algum erro ocorra o valor retornado deve ser zero. A busca pela palavra no texto deve iniciar na posição passada como parâmetro para a função.

    O tipo `texto` é dado por:

    ```
    const
       TAMMAX = 10000;

    type
       texto = array [1..TAMMAX] of char;
    ```

33. Escreva um programa em que gere e imprima um vetor de números reais de tamanho N, $1 \leq N \leq MAX$. A criação do vetor deve ser feita da seguinte maneira:

    - O tamanho N do vetor deve ser lido do teclado;
    - Os N números reais são gerados aleatoriamente no intervalo $[R_{min}, R_{max}[$, com R_{min} e R_{max} lidos do teclado;
    - A posição em que cada elemento real é inserida no vetor também é gerada aleatoriamente;
    - Se uma posição i sorteada já estiver ocupada, seu algoritmo deve encontrar a primeira posição j não ocupada, iniciando a partir de $i+1$ até o final do vetor. Se todas as posição entre $i+1$ e o final do vetor estiverem ocupadas, seu algoritmo deve pegar a primeira posição livre a partir do início do vetor.

    Dica: a função `random` sem parâmetros retorna um número real no intervalo $[0, 1[$, e a função `random(n)` retorna um número inteiro no intervalo $[0, n[$.

34. Escreva um procedimento em que remove um elemento de uma determinada posição p de um vetor v de n números reais. O vetor não está ordenado. Use a seguinte assinatura para o procedimento:

    ```
    procedure remove(var v: vetor; var n: integer; p: integer);
    ```

35. Escreva um procedimento em que altere um vetor de N números reais da seguinte forma: todos os elementos repetidos do vetor devem ir para o final do vetor, mas de maneira que estes últimos fiquem em ordem crescente. Exemplos:

```
ENTRADA: 5 3 8 2 3 9 8 9 7 5 3          ENTRADA: 4 4 3 3 2 2
SAÍDA  : 5 3 8 2 9 7 3 3 5 8 9          SAÍDA  : 4 3 2 2 3 4
```

36. Em uma festa estiveram presentes 150 pessoas. Cada uma delas recebeu um crachá na entrada com um número entre 1 e 150, número que representa a ordem de entrada de cada convidado.

    Como em toda festa, cada um dos presentes cumprimentou outras pessoas com apertos de mão. Ao final da festa, cada convidado sabia exatamente quantas vezes tinha apertado a mão de outras pessoas.

    Na saída, ao entregar o crachá ao recepcionista, cada convidado informou o número do seu crachá e quantas vezes trocou apertos de mão na festa.

    Muito curioso, o recepcionista queria saber quantos convidados eram muito populares no encontro, isto é, queria saber o número de pessoas que apertaram a mão de pelo menos outros 120 convidados.

    Faça um programa que modele o problema do recepcionista e que produza como saída o número de celebridades (cumprimentadas pelo menos 120 vezes) presentes na festa.

37. Um procedimento chamado `nova_geracao` recebe como parâmetros dois vetores (origem e destino) e o tamanho dos vetores. Este procedimento constrói um novo vetor de valores 0 ou 1 (destino) a partir do conteúdo do primeiro vetor (origem). Seja O o vetor origem e D o vetor destino, a regra para a construção do novo vetor é dada por:

    - se: $O[i-1] = 0,\ O[i] = 0,\ O[i+1] = 0$ então: $D[i] = 0$
    - se: $O[i-1] = 0,\ O[i] = 0,\ O[i+1] = 1$ então: $D[i] = 1$
    - se: $O[i-1] = 0,\ O[i] = 1,\ O[i+1] = 0$ então: $D[i] = 1$
    - se: $O[i-1] = 0,\ O[i] = 1,\ O[i+1] = 1$ então: $D[i] = 1$
    - se: $O[i-1] = 1,\ O[i] = 0,\ O[i+1] = 0$ então: $D[i] = 1$
    - se: $O[i-1] = 1,\ O[i] = 0,\ O[i+1] = 1$ então: $D[i] = 0$
    - se: $O[i-1] = 1,\ O[i] = 1,\ O[i+1] = 0$ então: $D[i] = 0$
    - se: $O[i-1] = 1,\ O[i] = 1,\ O[i+1] = 1$ então: $D[i] = 0$

    Onde i indica uma posição do vetor D. Considere o valor 0 para as bordas externas do vetor origem O. Escreva o procedimento `nova_geracao` e separe a regra descrita acima em uma função que dados os valores das 3 posições consecutivas do vetor origem $(O[i-1], O[i], O[i+1])$ calcula o valor correspondente no vetor destino $(D[i])$.

38. Faça um programa em que simule o tráfego em um trecho de uma rodovia de mão única, ou seja, uma rodovia na qual os veículos entram de um lado e saem do outro.

    - A rodovia é representada por um vetor com `TAM_RODOVIA` posições;
    - A simulação ocorre durante `MAX_TEMPO` iterações;

- Através da chamada do procedimento
  `detecta_entrada(VAR tipo, placa, velocidade:INTEGER)`,
  o programador é informado sobre a ocorrência ou não da entrada de um veículo na rodovia, bem como o tipo do veículo, sua placa e sua respectiva velocidade, onde:
  - *tipo:* 0 - nenhuma nova entrada, 1 - entrou automóvel, 2 - entrou caminhão;
  - *placa:* um número inteiro;
  - *velocidade:* a velocidade de deslocamento do veículo (em posições/unidade de tempo).
- Veículos do tipo automóvel ocupam uma posição da rodovia. Caminhões ocupam duas posições.
- Quando veículos mais rápidos alcançam veículos mais lentos, os primeiros devem andar mais devagar, pois não podem ultrapassar.

A cada unidade de tempo em que algum veículo sair da rodovia, seu programa deve imprimir esta unidade de tempo e o número da placa do veículo que saiu.

Exemplo: (`TAM_RODOVIA=7`, `MAX_TEMPO=10`)

- Entrada:
  - **t=1:** $tipo = 2, placa = 35, velocidade = 1$
  - **t=2:** $tipo = 0$
  - **t=3:** $tipo = 1, placa = 27, velocidade = 4$
  - **t=4:** $tipo = 0$
  - **t=5:** $tipo = 0$
  - **t=6:** $tipo = 1, placa = 16, velocidade = 2$
  - **t=7:** $tipo = 0$
  - **t=8:** $tipo = 0$
  - **t=9:** $tipo = 0$
  - **t=10:** $tipo = 0$
- Representação gráfica:
  - **t=1:**

35_1	35_1					

  - **t=2:**

	35_1	35_1				

  - **t=3:**

27_4		35_1	35_1			

  - **t=4:**

		27_4	35_1	35_1		

  - **t=5:**

			27_4	35_1	35_1	

  - **t=6:**

16_2				27_4	35_1	35_1

  - **t=7:**

		16_2			27_4	35_1

  - **t=8:**

				16_2		27_4

  - **t=9:**

						16_2

  - **t=10:**

- Saída:

  - **t=8:** 35
  - **t=9:** 27
  - **t=10:** 16

39. Você deve incluir no enunciado da questão anterior a existência de uma pista de ultrapassagem. Agora, veículos mais rápidos podem mover-se para a pista de ultrapassagem ao alcançarem veículos mais lentos, desde que não haja ninguém ocupando aquele trecho de pista. Eles devem retornar à pista original assim que tiverem completado a ultrapassagem, retomando a velocidade original. Você deve escrever apenas os procedimentos modificados ou novos que levam em conta este novo fato.

    Exemplo da nova saída para a entrada original:

    - Representação gráfica:
      - **t=1:**

35_1	35_1					

      - **t=2:**

	35_1	35_1				

      - **t=3:**

27_4		35_1	35_1			

      - **t=4:**

			27_4			
			35_1	35_1		

      - **t=5:**

				35_1	35_1	27_4

      - **t=6:**

16_2					35_1	35_1

      - **t=7:**

		16_2				35_1

      - **t=8:**

				16_2		

      - **t=9:**

						16_2

      - **t=10:**

    - Saída:
      - **t=6:** 27
      - **t=8:** 35
      - **t=10:** 16

40. Mateus, um engenheiro novato, está desenvolvendo uma notação posicional original para representação de números inteiros. Ele chamou esta notação de UMC (Um método curioso). A notação UMC usa os mesmos dígitos da notação decimal, isto é, de 0 a 9. Para converter um número A da notação UMC para a notação decimal deve-se adicionar K termos, onde K é o número de dígitos de A (na notação UMC). O

valor do i-ésimo termo correspondente ao i-ésimo dígito a_i, contando da direita para a esquerda é $a_i \times i!$.

Por exemplo, 719_{UMC} é equivalente a 53_{10}, pois $7 \times 3! + 1 \times 2! + 9 \times 1! = 53$.

Mateus está apenas começando seus estudos em teoria dos números e provavelmente não sabe quais as propriedades que um sistema de numeração deve ter, mas neste momento ele está apenas interessado em converter os números da notação UCM para a notação decimal. Você pode ajudá-lo?

Entrada: cada caso de teste é fornecido em uma linha simples que contém um número não vazio de no máximo 5 dígitos, representando um número em notação UMC. Este número não contém zeros a esquerda. O último teste é sequido por uma linha contendo um zero.

Saída: para cada caso de teste imprimir uma linha simples contendo a representação em notação decimal do correspondente número em UMC seguido do cálculo feito para a conversão.

O programa: seu programa deve, para cada número da entrada, convertê-lo em um vetor de inteiros, sendo que cada dígito do número é um elemento do vetor, e fazer os cálculos usando este vetor.

Exemplos de entrada e saída:

```
ENTRADA                 SAÍDA
719                     53 = 7 x 3! + 1 x 2! + 9 x 1!
1                       1 = 1 x 1!
15                      7 = 1 x 2! + 5 x 1!
110                     8 = 1 x 3! + 1 x 2! + 0 x 1!
102                     8 = 1 x 3! + 0 x 2! + 2 x 1!
0
```

41. Sabemos que nos compiladores mais recentes, nos quais existe o tipo string, podemos realizar de maneira simples operações com palavras. Imagine, no entanto, que estamos usando um compilador *Pascal* no qual não existe este tipo. Neste caso o programador deve implementar por sua própria conta os procedimentos com palavras. Neste exercício iremos considerar a seguinte declaração alternativa para o tipo string:

```
type palavra = array[1..MaxTam] of char;
```

Implemente uma função em *Pascal* que receba como parâmetros duas variáveis do tipo MeuString e retorne -1 se a primeira palavra for lexicograficamente menor que a segunda, 0 se forem iguais, e +1 no caso que resta.

42. Faça um programa que leia um certo número indefinido de vetores e que imprima o vetor original (O) e um vetor gerado (G) após um processo de compactação que consiste na eliminação de todos os elementos repetidos em cada vetor. Considere que a entrada de dados é feita em um vetor por linha, sendo que o primeiro elemento da linha é o tamanho de cada vetor e os elementos restantes da linha são os elementos do vetor. Quando o tamanho for zero significa que terminou a entrada de dados. Por exemplo, considere a seguinte entrada:

```
5 2 4 7 -1 2
3 1 1 1
7 3 4 5 3 4 5 1
0
```

Deverá produzir como saída o seguinte:

```
O: 2 4 7 -1 2
G: 2 4 7 -1
O: 1 1 1
G: 1
O: 3 4 5 3 4 5 1
G: 3 4 5 1
```

43. Considere uma sequência de dígitos binários como:

    011100011

    Uma maneira de criptografar essa sequência de bits é adicionar à cada dígito a soma dos seus dígitos adjacentes. Por exemplo, a sequência acima se tornaria:

    123210122

    Se P é a sequência original e Q é a sequência criptografada, então $Q[i] = P[i-1] + P[i] + P[i+1]$ para todas as posições i da sequência. Considerando uma sequência de tamanho n e seus índices variando de 0 a $n-1$, os dígitos $P[-1]$ e $P[n]$ não fazem parte da sequência original e são tratados como zeros na operação de codificação.

    Assumindo $P[0] = 0$ temos:

    - $Q[0] = P[0] + P[1] = 0 + P[1] = 1$, logo $P[1] = 1$.
    - $Q[1] = P[0] + P[1] + P[2] = 0 + 1 + P[2] = 2$, logo $P[2] = 1$.
    - $Q[2] = P[1] + P[2] + P[3] = 1 + 1 + P[3] = 3$, logo $P[3] = 1$.
    - Repetindo a operação temos: $P[4] = 0$, $P[5] = 0$, $P[6] = 0$, $P[7] = 1$ e $P[8] = 1$.

    Agora repetindo o mesmo processo para $P[0] = 1$ temos:

    - $Q[0] = P[0] + P[1] = 1 + P[1] = 1$, logo $P[1] = 0$.
    - $Q[1] = P[0] + P[1] + P[2] = 1 + 0 + P[2] = 2$, logo $P[2] = 1$.
    - $Q[2] = P[1] + P[2] + P[3] = 0 + 1 + P[3] = 3$, o que nos leva a conclusão que $P[3] = 2$. Entretanto isso viola o fato da sequência original ser binária. Portanto não existe uma decodificação possível considerando o primeiro dígito da sequência original valendo 1.

    Note que este algoritmo pode gerar ou decodificar uma sequência criptografada em até duas possíveis sequências originais, uma iniciando com 0 e outra iniciando com 1.

    Escreva um procedimento em que receba como parâmetros um vetor de números inteiros contendo a sequência criptografada e a decodifica em dois outros vetores de números inteiros. Caso uma das decodificações não seja possível, como no caso do

exemplo para $P[0] = 1$, o vetor correspondente deve ser preenchido com -1 na posição inicial.

Outros exemplos:

- $123210122 = 011100011, -1$
- $11 = 01, 10$
- $22111 = -1, 11001$
- $123210120 = -1, -1$
- $3 = -1, -1$
- $12221112222221112221111111112221111 =$
  $01101001101101001101001001001101001,$
  $10110010110110010110010010010110010$

44. Escrever um programa para ler um texto e imprimir uma distribuição de frequências para palavras do texto (quantas palavras de uma letra, quantas de duas letras, etc.).

45. Escreva um programa em *Pascal* que leia do teclado o gabarito de uma prova de 20 questões de múltipla escolha, onde as respostas são inteiros de 1 a 5. Em seguida, o programa deve ler o número de alunos que prestaram a prova e, para cada aluno, a sua matrícula (um inteiro) e as respectivas respostas. O programa deve calcular e escrever:

    - a relação de alunos ordenados pela nota, supondo que cada questão vale 5 pontos;
    - para cada questão: quantos alunos acertaram a questão

10.2 Matrizes

Assim como os vetores, as matrizes são *arrays*. Os vetores são estruturas unidimensionais enquanto que as matrizes são bidimensionais. Isto é, o acesso às posições de memória de um vetor são feitas com base em duas informações: o nome da variável e o deslocamento. Para se acessar os elementos de uma matriz, precisa-se do nome da variável, do deslocamento lateral e do deslocamento vertical. Os elementos de uma matriz também são do mesmo tipo.

10.2.1 Matrizes em *Pascal*

Para se declarar uma matriz de 200 posições inteiras, sendo 20 na vertical e 10 na horizontal, a linguagem *Pascal* usa a seguinte sintaxe (lembre-se que em outras linguagens a sintaxe pode ser diferente):

```
var m: array [1..20,1..10] of integer;
```

A construção "1..20,1..10" indica que existem 20 posições na horizontal (linhas) e 10 na vertical (colunas). O "of integer" indica que cada posição é para se guardar um número inteiro, isto é 2 bytes (dependendo da implementação).

Todas as restrições e variantes que usamos para os vetores valem também para as matrizes. Isto é, as declarações seguintes também são válidas:

```
var m: array [0..19,0..9] of integer;

var m: array [21..40,-19..-10] of integer;

var m: array [-19..0,51..60] of integer;

const maxLin=20, maxCol=10;
var m: array [1..maxLin,1..maxCol] of integer;
```

Agora, para escrever um valor qualquer, digamos 12345, na linha 15 e coluna 7 da matriz *m*, em *Pascal*, se usa um dos dois comandos seguintes:

```
m[15,7]:= 12345;

read(m[15,7]); (* e se digita 12345 no teclado *)
```

Por exemplo, a matriz abaixo tem 5 linhas e 4 colunas e pode ser visualizada do modo padrão:

	1	2	3	4
1	4	6	2	1
2	9	0	0	2
3	8	7	3	9
4	1	2	3	4
5	0	1	0	1

A declaração do tipo matriz em *Pascal* é assim:

```
type matriz= array [1..5,1..4] of integer;
```

No exemplo acima, a primeira linha é constituída pelos elementos seguintes: 4, 6, 2 e 1. A terceira coluna pelos elementos 2, 0, 3, 3 e 0. Podemos destacar a título de exemplo alguns elementos da matriz. Consideremos uma variável m do tipo matriz. Então: $m[3,2] = 7$, $m[2,3] = 0$, $m[3,4] = 9$, e assim por diante.

Notem que, isoladamente, cada linha ou coluna completa pode ser imaginada como sendo um vetor. De fato, uma outra maneira de se ver uma matriz é como sendo um vetor de vetores! Confira a declaração seguinte:

```
type vetor= array [1..4] of integer;
     matriz= array [1..5] of vetor;
var m: matriz;
```

Em *Pascal*, é correto dizer que, para o exemplo acima: $m[3][2] = 7$, $m[2][3] = 0$, $m[3][4] = 9$, e assim por diante. Nós usaremos a construção apresentada inicialmente.

10.2.2 Exemplos elementares

Do mesmo modo como fizemos para os vetores, vamos iniciar o estudo das matrizes apresentando problemas simples.

Lendo e imprimindo matrizes

A leitura dos elementos de uma matriz é bastante parecida com a leitura de vetores. Imaginando que cada linha da matriz é um vetor, basta fixar uma linha e ler todos os elementos das colunas. Agora é só repetir o procedimento para todas as linhas. Isto leva naturalmente a um código com um laço duplo: um para variar as linhas o outro para as colunas. A figura 10.38 permite visualizar o código para a leitura de uma matriz.

```
program ler_matriz;
var w: array [0..50,1..10] of real;
    i, j: integer;

begin
     for i:= 0 to 50 do
          for j:= 1 to 10 do
               read (w[i,j]);
end.
```

Figura 10.38: Lendo uma matriz.

Da forma como foi construído, este código exige que se digite os elementos da primeira linha, depois os da segunda e assim por diante. No laço mais interno, controlado pelo j, observem que o i é fixo, apenas o j varia, desta forma, a leitura das

colunas de uma linha fixa é idêntico a ler um vetor. O laço mais externo, controlado pelo i, faz variar todas as linhas.

Da mesma forma como apresentamos os vetores, vamos mostrar os códigos apenas em termos de procedimentos e funções. Para isto vamos precisar considerar as seguintes definições:

```
const maxLin = 50; maxCol = 40;
type matriz= array [0..maxLin,1..maxCol] of integer;
```

Também vamos considerar que apenas uma parte da matriz será usada, logo precisaremos sempre fazer a leitura das dimensões de uma matriz. Neste sentido, a figura 10.39 apresenta um procedimento que lê uma matriz de dimensões $n \times m$.

```
procedure ler_matriz (var w: matriz; var n,m: integer);
var i,j: integer;
begin
     read (n); (* n deve estar no intervalo 1..maxLin *)
     read (m); (* m deve estar no intervalo 1..maxLin *)

     for i:= 1 to n do
          for j:= 1 to m do
               read (w[i,j]);
end;
```

Figura 10.39: Procedimento para ler uma matriz.

As mesmas observações sobre passagem de parâmetros que foram feitas sobre os vetores se aplicam aqui, isto é, pelo overhead, sempre passamos matrizes por referência. A figura 10.40 apresenta um procedimento para impressão de uma matriz usando passagem de parâmetros por referência, embora não fosse estritamente necessário. O procedimento foi construído para imprimir a matriz linha por linha.

```
procedure imprimir_matriz (var w: matriz; n,m: integer);
var i,j: integer;
begin
     for i:= 1 to n do
     begin
          for j:= 1 to m do
               write (w[i,j],' ');
          writeln; (* muda de linha a cada fim de coluna *)
     end;
end;
```

Figura 10.40: Procedimento para imprimir uma matriz.

Tomando como exemplo a matriz do início desta seção, se forem digitados todos os valores, linha por linha, para cada uma do início até o final de cada coluna, teríamos em memória algo como ilustrado na figura seguinte:

	1	2	3	4	5	...	40
1	4	6	2	1	?		?
2	9	0	0	2	?		?
3	8	7	3	9	?		?
4	1	2	3	4	?		?
5	0	1	0	1	?		?
⋮							
50	?	?	?	?	?		?

A título de exemplo, poderíamos construir um procedimento que imprime a matriz em sua forma transposta, isto é, com as linhas e colunas invertidas. Isto é apresentado na figura 10.41. Basta inverter i e j no comando de impressão! Observem que a matriz não mudou na memória, apenas a impressão é diferente.

```
procedure imprimir_transposta (var w: matriz; n,m: integer);
var i,j: integer;
begin
      for i:= 1 to m do
      begin
            for j:= 1 to n do
                  write (w[j,i],' ');
            writeln;
      end;
end;
```

Figura 10.41: Procedimento para imprimir a transposta de uma matriz.

Outros procedimentos interessantes são os de impressão de apenas uma certa linha (figura 10.42) ou de apenas uma certa coluna (figura 10.43).

```
procedure imprimir_uma_linha (var w: matriz; n,m: integer; K: integer);
(* imprime a linha K da matriz *)
var j: integer;
begin
      for j:= 1 to m do
            write (w[K,j],' '); (* K fixo na primeira posicao *)
      writeln;
end;
```

Figura 10.42: Procedimento para imprimir uma unica linha da matriz.

Considerando o exemplo acima e fazendo $K = 2$ teríamos a seguinte saída:

```
9 0 0 2
```

Considerando o exemplo acima e fazendo $K = 2$ teríamos a seguinte saída:

```
procedure imprimir_uma_coluna(var w: matriz; n,m: integer; K: integer);
(* imprime a coluna K da matriz *)
var i: integer;
begin
      for i:= 1 to n do
            writeln (w[i,K]); (* K fixo na segunda posicao *)
end;
```

Figura 10.43: Procedimento para imprimir uma unica coluna da matriz.

```
6
0
7
2
1
```

Outro exemplo interessante seria imprimir apenas os elementos da matriz que são pares, conforme mostrado na figura 10.44.

```
procedure imprimir_os_pares(var w: matriz; n,m: integer);
var i,j : integer;
begin
      for i:= 1 to n do
      begin
            for j:= 1 to m do
                  if eh_par (w[i,j]) then
                        write (w[i,j],' ');
      end;
      writeln;
end;
```

Figura 10.44: Procedimento para imprimir os elementos pares matriz.

Considerando novamente a nossa matriz exemplo, teríamos a seguinte saída:

```
4 6 2 0 0 2 8 2 4 0 0
```

Para finalizarmos esta seção inicial, apresentamos na figura 10.45 um código que imprime os elementos cujos índices das linhas e das colunas são pares. Considerando novamente a nossa matriz exemplo, teríamos a seguinte saída:

```
0 2
2 4
```

```
procedure imprimir_as_linhas_e_colunas_pares(var w: matriz; n,m: integer);
var i,j : integer;
begin
      for i:= 1 to n do
      begin
            for j:= 1 to m do
                  if eh_par (i) and eh_par(j) then
                        write (w[i,j],' ');
            writeln;
      end;
end;
```

Figura 10.45: Procedimento para imprimir os elementos cujos indices são pares.

Encontrando o menor elemento de uma matriz

Vamos retornar ao velho e conhecido problema de se determinar qual é o menor elemento de um conjunto de números em memória, considerando que, desta vez, eles estarão armazenados em uma matriz. A figura 10.46 contém o código que faz isto. Note a semelhança com os programas das seções anteriores. A técnica é a mesma: o primeiro elemento é considerado o menor de todos e depois a matriz tem que ser toda percorrida (todas as linhas e todas as colunas) para ver se tem outro elemento ainda menor.

```
function acha_menor_matriz (var w: matriz; n,m: integer): integer;
var i,j: integer;
    menor: integer;
begin
      menor:= w[1,1];
      for i:= 1 to n do
            for j:= 1 to m do
                  if w[i,j] < menor then
                        menor:= w[i,j];
      acha_menor_matriz:= menor;
end;
```

Figura 10.46: Encontrando o menor elemento de uma matriz.

Soma de matrizes

Nesta seção vamos implementar o algoritmo que soma duas matrizes. Para isto precisamos antes entender como funciona o processo.

Sejam v e w duas matrizes. Para somá-las, é preciso que elas tenham o mesmo tamanho. Isto posto, o algoritmo cria uma nova matriz $v + w$ onde cada elemento i, j da nova matriz é a soma dos respectivos elementos $v[i, j]$ e $w[i, j]$. O esquema é muito parecido com a soma de vetores, já estudada, apenas, novamente, trata-se também da segunda dimensão.

Desta forma, o algoritmo que soma os dois vetores deverá, para cada par i, j fixo, somar os respectivos elementos em v e w e guardar em $v + w$. Variando i de 1 até o número de linhas e j de 1 até o número de colunas resolve o problema. O programa que implementa esta ideia é apresentado na figura 10.47.

```
procedure somar_matrizes (var v, w, soma_v_w: matriz; n,m: integer);
var i,j: integer;
begin
     (* n e m sao o numero de linhas e colunas, respectivamente *)
     for i:= 1 to n do
          for j:= 1 to m do
               soma_v_w[i,j]:= v[i,j] + w[i,j];
end;
```

Figura 10.47: Somando duas matrizes.

É interessante comparar com o procedimento que soma vetores, apresentado na figura 10.13. Se considerarmos que, no laço interno controlado pelo j o i é fixo, então os dois procedimentos fazem a mesma operação de somar dois vetores!

Multiplicação de matrizes

Agora vamos resolver o problema da multiplicação de matrizes. Inicialmente vamos recordar como isto é feito.

Do ponto de vista matemático cada elemento da matriz resultado da multiplicação de duas matrizes pode ser encarado como sendo o produto escalar de dois vetores formados por uma linha da primeira matriz e por uma coluna da segunda.

Vamos considerar duas matrizes $A_{n\times m}$ e $B_{m\times p}$. A multiplicação só pode ocorrer se o número de colunas da matriz A for igual ao número de linhas da matriz B. Isto por causa do produto escalar de vetores, que exige que os vetores tenham o mesmo tamanho. O resultado é uma matriz $n \times p$. O produto escalar de vetores, conforme já estudado, é o resultado da seguinte somatória:

$$\sum_{k=1}^{m} V[k] \times W[k].$$

Vamos considerar duas matrizes A e B, fixando uma linha I na matriz A e uma coluna J na matriz B. Conforme já mencionado, fixar linhas ou colunas em matrizes é o mesmo que trabalhar com vetores. Então, neste caso, o produto escalar da linha I da matriz A pela coluna J da matriz B é dado pela seguinte somatória:

$$\sum_{k=1}^{m} A[I, k] \times B[k, J].$$

O programa que realiza esta operação é uma adaptação simples do código exibido na figura 10.14, que para fins didáticos é mostrado na figura 10.48.

```
begin  (* considerando I e J fixos *)
      soma:= 0;
      for i:= 1 to m do
            soma:= soma + A[I,k] * B[k,J];
      prod_escalar:= soma;
end;
```

Figura 10.48: Produto escalar de uma linha da matriz por uma coluna da outra.

Vejamos isto de forma ilustrada considerando as duas matrizes seguintes:

$$A: \begin{array}{cccc} 4 & 6 & 2 & 1 \\ 9 & 0 & 0 & 2 \\ \boxed{8} & \boxed{7} & \boxed{3} & \boxed{9} \\ 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 1 \end{array} \qquad B: \begin{array}{ccc} 2 & \boxed{3} & 4 \\ 5 & \boxed{0} & 0 \\ 0 & \boxed{7} & 7 \\ 1 & \boxed{0} & 9 \end{array}$$

O produto escalar da linha (terceira) pela coluna (segunda) em destaque é o resultado da seguinte operação: $8 \times 3 + 7 \times 0 + 3 \times 7 + 9 \times 0 = 45$. Este resultado é o valor da linha 3 coluna 2 da matriz produto, isto é, é o elemento AB[3,2].

Para se obter a resultado completo do produto de A por B, basta variar I nas linhas de A e J nas colunas de B. Isto é apresentado na figura 10.49.

```
procedure multiplicar_matrizes (var A: matriz;       lin_A, col_A: integer;
                                var B: matriz;       lin_B, col_B: integer;
                                var AB: matriz; var lin_AB, col_AB: integer);
var i, j, k: integer;
begin
      lin_AB:= lin_A; col_AB:= col_B;
      for i:= 1 to lin_A do
            for j:= 1 to col_B do
            begin
                  AB[i,j]:= 0;
                  for k:= 1 to lin_B do
                        AB[i,j]:= AB[i,j] + A[i,k] * B[k,j];
            end;
end;
```

Figura 10.49: Multiplicação de duas matrizes.

10.2.3 Procurando elementos em matrizes

Nesta seção vamos apresentar alguns problemas de busca de elementos em matrizes.

Busca em uma matriz

O problema de busca em matrizes é similar ao caso dos vetores. O procedimento agora é quadrático, pois no pior caso a matriz inteira deve ser percorrida (caso em que o elemento não está na matriz). A figura 10.50 contém o código para este problema.

```
function busca (var w: matriz; n,m: integer; x: integer): boolean;
var i,j: integer; achou: boolean;
begin
      achou:= false;
      i:= 1;
      while (i <= n) and not achou do
      begin
            j:= 1;
            while (j <= m) and not achou do
            begin
                  if w[i,j] = x then achou:= true;
                  j:= j + 1;
            end;
            i:= i + 1;
      end;
      busca:= achou;
end;
```

Figura 10.50: Busca em uma matriz.

As vezes é preciso saber as coordenadas i e j do elemento. A figura 10.51 mostra como adaptar a função anterior com duas modificações: a primeira é que deve-se usar um procedimento que retorna as duas coordenadas usando parâmetros por referência, pois funções retornam apenas um único valor. A outra diferença é que, quando o elemento é encontrado deve-se lembrar da linha e coluna correspondente.

Um problema interessante é saber se uma matriz contém elementos repetidos. Basta varrer a matriz e, para cada elemento, saber se ele existe na matriz em posição diferente. Isto exige um aninhamento de quatro laços!

Um laço duplo é necessário para se percorrer a matriz por completo, e depois um outro laço duplo para cada elemento para se saber se ele se repete. Isto resulta em um algoritmo de ordem de n^4 para uma matriz quadrada de ordem n, para o pior caso. Para completar o grau de dificuldade, queremos parar o processamento tão logo um elemento repetido seja encontrado. O código final está ilustrado na figura 10.52.

10.2.4 Inserindo uma coluna em uma matriz

Vamos considerar o problema de receber uma matriz de dimensões $n \times m$ e um vetor de n elementos e inserir o vetor como uma coluna adicional na matriz, que ficará com dimensões $n \times m + 1$.

Por exemplo, consideremos a nossa matriz exemplo e o seguinte vetor:

	1	2	3	4
1	4	6	2	1
2	9	0	0	2
3	8	7	3	9
4	1	2	3	4
5	0	1	0	1

1	2	3	4	5
7	6	5	4	3

```
function acha_pos_elemento (var w: matriz; n,m,x: integer; var l,c: integer): boolean;
var i,j: integer; achou: boolean;
begin
      achou:= false;
      i:= 1;
      while (i <= n) and not achou do
      begin
            j:= 1;
            while (j <= m) and not achou do
            begin
                  if w[i,j] = x then
                  begin
                        (* quando acha o elemento , armazena as coordenadas *)
                        achou:= true;
                        l:= i;
                        c:= j;
                  end;
                  j:= j + 1;
            end;
            i:= i + 1;
      end;
      acha_pos_elemento:= achou;
end;
```

Figura 10.51: Busca em uma matriz, retornando as coordenadas (l,c).

Inicialmente vamos apenas inserir o vetor após a última coluna, isto é, o vetor será a última coluna da nova matriz, tal como na figura seguinte, de ordem 5×5 (vetor inserido está em negrito na figura):

	1	2	3	4	5
1	4	6	2	1	**7**
2	9	0	0	2	**6**
3	8	7	3	9	**5**
4	1	2	3	4	**4**
5	0	1	0	1	**3**

O procedimento mostrado na figura 10.53 faz esta operação.

Um problema mais difícil seria se quiséssemos inserir o vetor em alguma coluna que não fosse a última da matriz. O exemplo seguinte mostra nossa matriz de exemplo com o vetor inserido na coluna 2.

```
function tem_repetidos (var w: matriz; n,m: integer): boolean;
var i,j, p, q: integer;
    repetiu: boolean;
begin
     repetiu:= false;
     i:= 1;
     while (i <= n) and not repetiu do
     begin
          j:= 1;
          while (j <= m) and not repetiu do
          begin
               p:= 1;
               while (p <= n) and not repetiu do
               begin
                    q:= 1;
                    while (q <= m) and not repetiu do
                    begin
                         if (w[p,q] = w[i,j]) and ((p <> i) or (q <> j)) then
                              repetiu:= true;
                         q:= q + 1;
                    end;
                    p:= p + 1;
               end;
               j:= j + 1;
          end;
          i:= i + 1;
     end;
     tem_repetidos:= repetiu;
end;
```

Figura 10.52: Verifica se uma matriz tem elementos repetidos.

	1	2	3	4	5
1	4	**7**	6	2	1
2	9	**6**	0	0	2
3	8	**5**	7	3	9
4	1	**4**	2	3	4
5	0	**3**	1	0	1

Neste caso, tal como no caso dos vetores, teríamos que abrir espaço na matriz antes de inserir o vetor. Esta é uma operação bastante custosa, pois temos que mover várias colunas para frente, cada uma delas move n elementos para frente. O algoritmo apresentado na figura 10.54 mostra estes dois passos, um que abre espaço o outro que insere o vetor no espaço aberto.

Para inserir linhas em uma matriz o procedimento é análogo.

```
procedure inserir_coluna_no_fim (var w: matriz; v: vetor_i; var n,m: integer);
(* recebe uma matriz e um vetor e insere o vetor como ultima coluna da matriz *)
var i: integer;

begin
     for i:= 1 to n do
          w[i,m+1] := v[i];   (* m+1 eh fixo, queremos sempre a ultima coluna *)
     m:= m + 1;               (* altera o numero de colunas                  *)
end;
```

Figura 10.53: Insere um vetor como última coluna de uma matriz.

```
procedure insere_coluna_k (var w: matriz; var v: vetor_i; var n,m: integer; K: integer)
    ;
(* recebe uma matriz e um vetor e insere o vetor na coluna K da matriz *)
var i,j: integer;

begin
     (* primeiro abre espaco *)
     for j:= m downto K do       (* para cada coluna, iniciando na ultima *)
          for i:= 1 to n do      (* move elementos uma coluna para frente *)
               w[i,j+1]:= w[i,j];

     (* depois insere na coluna K *)
     for i:= 1 to n do
          w[i,K] := v[i];   (* K eh fixo, queremos sempre a K-esima coluna *)
     m:= m + 1;             (* altera o numero de colunas                  *)
end;
```

Figura 10.54: Insere um vetor como K-ésima coluna de uma matriz.

10.2.5 Aplicações de matrizes em imagens

Nosso objetivo nesta seção é mostrar como podemos fazer modificações em imagens digitais no formato PGM. Antes vamos resolver dois problemas que serão úteis no decorrer deste capítulo.

Primeiro desafio

O primeiro desafio é, dada uma matriz de dimensões $n \times n$, vamos gerar uma segunda matriz a partir da primeira onde cada elemento é a média da soma dele com três de seus vizinhos na matriz original. Para exemplificar o que queremos, vejamos a seguinte ilustração de uma matriz 4×4:

$$\begin{matrix} 4 & 6 & 2 & 1 \\ 9 & 0 & 0 & 2 \\ 8 & 7 & 3 & 9 \\ 1 & 2 & 3 & 4 \end{matrix}$$

Queremos que a matriz nova tenha como elemento da primeira linha, primeira coluna, a média do quadrado constituído pelos elementos das duas primeiras linhas considerando-se apenas as duas primeiras colunas, isto é, a sub-matriz:

$$\begin{matrix} 4 & 6 \\ 9 & 0 \end{matrix}$$

A média destes elementos é $\frac{(4+6+9+0)}{4} = 4.75$.

O elemento gerado para a primeira linha, segunda coluna é a média da seguinte sub-matriz:

$$\begin{matrix} 2 & 1 \\ 0 & 2 \end{matrix}$$

Isto é, $\frac{(2+1+0+2)}{4} = 1.25$. E assim por diante. No final, queremos produzir a seguinte matriz, de ordem 2×2:

$$\begin{matrix} 4.75 & 1.25 \\ 4.50 & 4.75 \end{matrix}$$

O procedimento que realiza este cálculo é ilustrado na figura 10.60. Como estamos no início do estudo, vamos considerar sempre que a matriz as dimensões n e m são sempre pares.

```
procedure ler_pgm (var O: imagem; var l,c,max: integer);
var i,j: integer;
    s: string[2];
begin
        readln (s);
        if s = 'P2' then
        begin
                read (c,l);
                read (max);
                for i:= 1 to l do
                        for j:= 1 to c do
                                read (O[i,j]);
        end
        else
                writeln ('Formato invalido');
end;
```

Figura 10.55: Leitura de uma imagem PGM.

```
procedure imprimir_pgm (var O: imagem; l,c,max: integer);
var i,j: integer;
begin
        writeln ('P2');
        writeln (c,' ',l);
        writeln (max);
        for i:= 1 to l do
        begin
                for j:= 1 to c-1 do
                        write (O[i,j],' ');
                writeln (O[i,c]);
        end;
end;
```

Figura 10.56: Impressão de uma imagem PGM.

```
function maior_valor (var O: imagem; l,c: integer): integer;
var i,j, m: integer;
begin
        m:= O[1,1];
        for i:= 1 to l do
                for j:= 1 to c do
                        if O[i,j] > m then
                                m:= O[i,j];
        maior_valor:= m;
end;
```

Figura 10.57: Cálculo do valor do maior pixel.

Segundo desafio

Agora vamos trabalhar com geração de "pedaços" de uma matriz, isto é, dada uma matriz $n \times m$, queremos gerar uma nova matriz que consiste de uma submatriz da primeira.

Para isto, vamos considerar uma "janela" da matriz original definida pelo canto superior esquerdo e pelo canto inferior direito. Vejamos um exemplo do que queremos. Vamos considerar novamente a matriz exemplo seguinte, de ordem 5×4:

$$\begin{array}{cccc} 4 & 6 & 2 & 1 \\ 9 & 0 & \mathbf{0} & \mathbf{2} \\ 8 & 7 & \mathbf{3} & \mathbf{9} \\ 1 & 2 & \mathbf{3} & \mathbf{4} \\ 0 & 1 & 0 & 1 \end{array}$$

A janela exemplo tem seu canto superior esquerdo nas coordenadas $(2, 3)$ e seu canto inferior direito em $(4, 4)$. Isto define a submatriz seguinte (elementos em negrito na figura anterior):

```
procedure clarear_pgm (var O: imagem; l,c,max,cte: integer);
var i,j: integer;
begin
        for i:= 1 to l do
                for j:= 1 to c do
                begin
                        O[i,j]:= O[i,j] + cte;
                        if O[i,j] > max then
                                O[i,j]:= max;
                end;
end;
```

Figura 10.58: Procedure para clarear uma imagem PGM.

```
function media_4_vizinhos (var O: imagem; i,j: integer): integer;
var x,y: integer;
begin
        x:= 2*i - 1;
        y:= 2*j - 1;
        media_4_vizinhos:= (O[x,y] + O[x+1,y] + O[x,y+1] + O[x+1,y+1]) div 4;
end;
```

Figura 10.59: Função que calcula média dos quatro vizinhos de um pixel.

$$\begin{matrix} 0 & 2 \\ 3 & 9 \\ 3 & 4 \end{matrix}$$

O procedimento apresentado na figura 10.62 gera a submatriz desejada.

Matrizes que representam imagens

Um dos formatos reconhecidos pelos computadores atuais para imagens é o padrão PGM. Este formato consiste de um arquivo ASCII que tem o seguinte formato:

- a primeira linha contém um identificador "P2";
- a segunda linha contém a largura e a altura de uma matriz, isto é, o número de colunas e de linhas;
- a terceira linha contém o valor do maior valor da matriz que contém a imagem propriamente dita;
- o restante do arquivo contém uma matriz de elementos (bytes) que representam um pixel da imagem em tons de cinza.

A figura 10.63 mostra um exemplo de um arquivo que é uma imagem em PGM: a primeira linha tem "P2"; a segunda linha contém a dimensão da matriz (10×11,

```
procedure zoom_pgm (var O: imagem;  lO,cO: integer;
                    var D: imagem; var lD, cD, maxD: integer);
var i,j: integer;
begin
        lD:= lO div 2;
        cD:= cO div 2;
        for i:= 1 to lD do
                for j:= 1 to cD do
                        D[i,j]:= media_4_vizinhos (O,i,j);
        maxD:= maior_valor (D,lD,cD);
end;
```

Figura 10.60: Procedure para fazer zoom em uma imagem PGM.

observe que por definição o número de colunas vem antes); a terceira linha contém o maior elemento (40) da matriz que constitui o restante do arquivo.

Vamos mostrar uma maneira de se fazer um *zoom* na imagem. Existem várias técnicas, a nossa será da seguinte forma: o *zoom* será obtido pela média de cada quatro vizinhos, isto é, o procedimento da figura **??**.

A matriz que contém o *zoom* será então impressa na saída padrão, mas no formato correto para poder ser visualizada com qualquer aplicativo padrão para imagens. Por isto não podemos esquecer de, na saída padrão, acrescentar o cabeçalho do formato PGM, isto é, uma linha contendo "P2", outra contendo número de colunas e de linhas e o valor do maior pixel.

Notem que o procedimento para se achar o maior pode ser facilmente adaptado do programa da figura 10.46, inclusive para se retornar um valor do tipo *byte* e não do tipo *real*.

Interessante observar que este zoom pode ser de vários níveis, bastando para isto adaptar-se o procedimento da média no que gera as submatrizes. Mas isto fica como exercício. O programa para o *zoom* está mostrado na figura 10.64.

```
procedure detectar_bordas_pgm (var O,D: imagem; l,c: integer; var max: integer);
var i,j,grad: integer;
begin
        (* bordas recebem zero *)
        for i:= 1 to l do
        begin
                D[i,1]:= 0;
                D[i,c]:= 0;
        end;
        for i:= 1 to c do
        begin
                D[1,i]:= 0;
                D[l,i]:= 0;
        end;

        for i:= 2 to l-1 do
                for j:= 2 to c-1 do
                begin
                        grad:= abs (O[i,j]*4 -
                        (O[i-1,j] + O[i+1,j] +
                        O[i,j-1] + O[i,j+1]));
                        if grad > limiar then
                           D[i,j]:= 255
                        else
                           D[i,j]:= 0;
                end;
        max:= 255;
end;
```

Figura 10.61: Procedure para detectar bordas de uma imagem PGM.

```
procedure gerasubmatriz(var m: matriz;      lin_m, col_m: integer;
                        var w: matriz; var lin_w, col_w: integer;
                        x1,y1,x2,y2: integer);
(* (x1,y1) eh o canto superior esquerdo, (x2,y2) eh o canto inferior esquerdo *)
var i,j : integer;
begin
  lin_w:= x2 - x1 + 1;
  col_w:= y2 - y1 + 1;

  for i:= 1 to lin_w do
      for j:= 1 to col_w do
         w[i,j]:= m[i+x1-1,j+y1-1];
end;
```

Figura 10.62: Gerando submatriz.

```
P2
11 10
40
40 5  5  5  5  5  5  5  5  40 0
5  20 20 5  5  5  5  5  5  5  5
5  5  20 5  5  5  0  0  0  0  0
5  5  20 20 5  5  20 20 0  0  5
5  5  5  5  5  5  0  20 0  0  0
5  5  5  5  5  5  0  20 20 0  5
5  5  5  5  11 11 11 0  0  0  0
5  5  5  5  20 20 11 5  5  5  5
5  5  5  5  11 20 11 5  5  5  0
40 5  5  5  11 20 20 5  5  40 5
```

Figura 10.63: Exemplo de imagem no formato PGM.

```
program pgm;

const X = 50;
      limiar = 50;

type imagem = array [1..300,1..300] of integer;

var O,D: imagem;
    linO,colO,maxO,linD,colD,maxD: integer;

{$I procedure_ler_pgm.pas}
{$I procedure_imprimir_pgm.pas}
{$I function_maior_valor_pgm.pas}
{$I procedure_clarear_pgm.pas}
{$I function_media_4_vizinhos_pgm.pas}
{$I procedure_zoom_pgm.pas}
{$I procedure_detectar_bordas_pgm.pas}

begin
  ler_pgm (O,linO,colO,maxO);
  clarear_pgm (O,linO,colO,maxO,X);
  imprimir_pgm (O,linO,colO,maxO);
  zoom_pgm (O,linO,colO,D,linD,colD,maxD);
  detectar_bordas_pgm (O,D,linO,colO,maxD);
  imprimir_pgm (D,linO,colO,maxD);
end.
```

Figura 10.64: Programa que le imagem PGM e gera imagem com *zoom*.

10.2.6 Exercícios

1. Dada uma matriz real A com m linhas e n colunas e um vetor real V com n elementos, determinar o produto de A por V.

2. Um vetor real X com n elementos é apresentado como resultado de um sistema de equações lineares $Ax = B$ cujos coeficientes são representados em uma matriz real $A(m \times n)$ e os lados direitos das equações em um vetor real B de m elementos. Verificar se o vetor X é realmente solução do sistema dado.

3. Dizemos que uma matriz inteira $A(n \times n)$ é uma matriz de permutação se em cada linha e em cada coluna houver $n - 1$ elementos nulos e um único elemento igual a 1. Dada uma matriz inteira $A(n \times n)$ verificar se A é de permutação. Exemplos:

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

   é de permutação, enquanto que esta outra não é:

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{matrix}$$

4. Dada uma matriz $A(n \times m)$ imprimir o número de linhas e o número de colunas nulas da matriz. Exemplo: a matriz abaixo tem duas linhas e uma coluna nulas.

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 2 \\ 4 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 \end{matrix}$$

5. Dizemos que uma matriz quadrada inteira é um quadrado mágico se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos das diagonais principal e secundária são todos iguais. Exemplo:

$$\begin{matrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{matrix}$$

   é um quadrado mágico pois 8+0+7 = 4+5+6 = 3+10+2 = 8+4+3 = 0+5+10 = 7+6+2 = 8+5+2 = 3+5+7 = 15.

(a) Dada uma matriz quadrada $A(n \times m)$, verificar se A é um quadrado mágico.

(b) Informe quantas são e quais são as submatrizes não triviais (isto é, não pode ser a matriz constituída por apenas um elemento, uma linha e uma coluna) que definem quadrados mágicos. Por exemplo, a matriz do exemplo acima tem 4 submatrizes de tamanho 2×2, duas submatrizes de tamanho 2×3, etc, e uma única submatriz de dimensão 3×3;

- Armazene de alguma maneira as informações necessárias sobre a localização precisa de cada uma das submatrizes não triviais que definem quadrados mágicos;
- Imprima a qualquer tempo algum quadrado mágico armazenado;
- Dado uma dimensão qualquer, digamos N, imprima todas os quadrados mágicos de dimensão N contidos na matriz original.

6. Implemente o quadrado "quase magico". Um quadrado quase magico é aquele em que as somas das linhas e a somas das colunas resultam em um mesmo valor, mas a soma dos elementos das diagonais não. O programa deve pedir a dimensão do quadrado a ser impresso, que deve ser um número ímpar entre 1 e 99.

7. Um jogo de palavras cruzadas pode ser representado por uma matriz $A(n \times m)$ onde cada posição da matriz corresonde a um quadrado do jogo, sendo que 0 indica um quadrado em branco e -1 indica um quadrado preto. Colocar as numerações de início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras.

   Exemplo: Dada a matriz:

0	-1	0	-1	-1	0	-1	0
0	0	0	0	-1	0	0	0
0	0	-1	-1	0	0	-1	0
-1	0	0	0	0	-1	0	0
0	0	-1	0	0	0	-1	-1

   A saída deveria ser:

1	-1	2	-1	-1	3	-1	4
5	6	0	0	-1	7	0	0
8	0	-1	-1	9	0	-1	0
-1	10	0	11	0	-1	12	0
13	0	-1	14	0	0	-1	-1

8. Uma matriz $D(8 \times 8)$ pode representar a posição atual de um jogo de damas, sendo que 0 indica uma casa vazia, 1 indica uma casa ocupada por uma peça branca e -1 indica uma casa ocupada por uma peça preta. Supondo que as peças pretas estão se movendo no sentido crescente das linhas da matriz D, determinar as posições das peças pretas que:

- podem tomar peças brancas;
- podem mover-se sem tomar peças brancas;
- não podem se mover.

9. Deseja-se atualizar as contas correntes dos clientes de uma agência bancária. É dado o cadastro de N clientes contendo para cada cliente o número de sua conta e seu saldo. O cadastro está ordenado pelo número da conta. Em seguida é dado o número de operações realizadas no dia, e, para cada operação, o número da conta, uma letra C ou D indicando se a operação é de crédito ou débido, e o valor da operação. Emitir o cadastro de clientes atualizado. Pode ser modelado como uma matriz $N \times 2$.

10. Reordenar a matriz do exercício anterior por ordem de saldo, do maior para o menor.

11. Os elementos $M[i, j]$ de uma matriz $M(n \times n)$ representam os custos de transporte da cidade i para a cidade j. Dados n itinerários lidos do teclado, cada um com k cidades, calcular o custo total para cada itinerário. Exemplo:

```
4 1 2 3
5 2 1 400
2 1 3 8
7 1 2 5
```

O custo do itinerário 1 4 2 4 4 3 2 1 é: M[1,4] + M[4,2] + M[2,4] + M[4,4] + M[4,3] + M[3,2] + M[2,1] = 3 + 1 + 400 + 5 + 2 + 1 + 5 = 417.

12. Considere n cidades numeradas de 1 a n que estão interligadas por uma série de estradas de mão única. As ligações entre as cidades são representadas pelos elementos de uma matriz quadrada $L(n \times n)$ cujos elementos L$[i, j]$ assumem o valor 0 ou 1 conforme exista ou não estrada direta que saia da cidade i e chegue na cidade j. Assim, os elementos da i-ésima linha indicam as estradas que saem da cidade i e os elementos da j-ésima coluna indicam as estradas que chegam à cidade j. Por convenção, $L[i, i] = 1$. A figura abaixo ilustra um exemplo para $n = 4$.

	A	B	C	D
A	1	1	1	0
B	0	1	1	0
C	1	0	1	1
D	0	0	1	1

Por exemplo, existe um caminho direto de A para B mas não de A para D.

(a) Dado k, determinar quantas estradas saem e quantas chegam à cidade k.

(b) A qual das cidades chega o maior número de estradas?

(c) Dado k, verificar se todas as ligações diretas entre a cidade k e outras são de mão dupla;

(d) Relacionar as cidades que possuem saídas diretas para a cidade k;

(e) Relacionar, se existirem:

- As cidades isoladas, isto é, as que não têm ligação com nenhuma outra;
- As cidades das quais não há saída, apesar de haver entrada;
- As cidades das quais há saída sem haver entrada;

(f) Dada uma sequência de m inteiros cujos valores estão entre 1 e n, verificar se é possível realizar o roteiro correspondente. No exemplo dado, o roteiro representado pela sequência ($m = 5$) 3 4 3 2 1 é impossível;

(g) Dados k e p, determinar se é possível ir da cidade k até a cidade p pelas estradas existentes. Você consegue encontrar o menor caminho entre as duas cidades?

(h) Dado k, dterminar se é possível, partindo de k, passar por todas as outras cidades apenas uma vez e retornar a k.

13. Uma matriz transposta M^T é o resultado da troca de linhas por colunas em uma determinad a matriz M. Escreva um programa que leia duas matrizes (A e B), e testa se $B = A + A^T$.

14. Fazer procedimentos que recebam três parâmetros: uma matriz e dois inteiros representando as dimensões da matriz. Retornar:

(a) a transposta de matriz;

(b) a soma das duas matrizes;

(c) a multiplicação das duas matrizes.

15. Faça um programa que leia duas matrizes A e B quaisquer e imprima a transposta de B se a transposta de A for igual a B.

16. Fazer uma função que receba como parâmetros: dois vetores de reais e dois inteiros representando as dimensões dos vetores. Retornar o produto escalar dos dois vetores (real). Refazer a multiplicação de matrizes usando esta função.

17. Faça um programa que, dadas N datas em uma matriz $\text{DATAS}_{N\times 3}$, onde a primeira coluna corresponde ao dia, a segunda ao mês e a terceira ao ano, coloque essas datas em ordem cronológica crescente. Por exemplo:

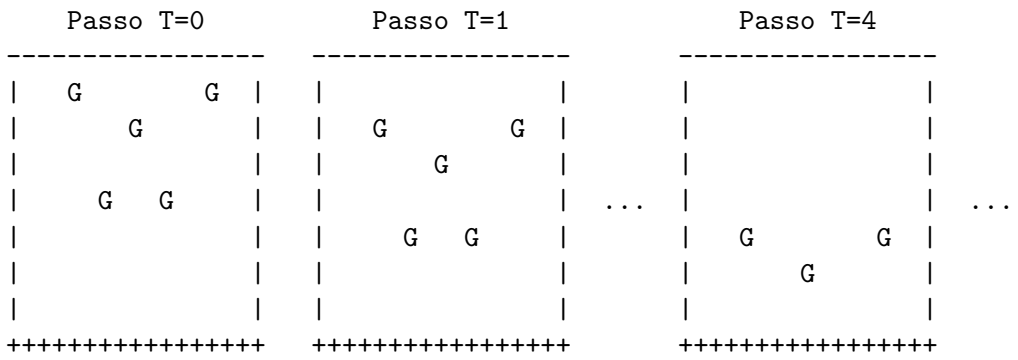
$$DATAS = \begin{pmatrix} 5 & 1 & 1996 \\ 25 & 6 & 1965 \\ 16 & 3 & 1951 \\ 15 & 1 & 1996 \\ 5 & 11 & 1965 \end{pmatrix} DATAS = \begin{pmatrix} 16 & 3 & 1951 \\ 25 & 6 & 1965 \\ 5 & 11 & 1965 \\ 5 & 1 & 1996 \\ 15 & 1 & 1996 \end{pmatrix}$$

18. Verifique se a matriz A é simétrica, isto é, se $A[i,j] = A[j,i], \forall i,j \leq M$. Faça uma função que retorne 1 em caso afirmativo, 0 caso contrário.

19. Uma matriz B é dita inversa da matriz A quando $A \times B = I$, onde I é a matriz identidade e $\times$ é a operação de multiplicação de matrizes. A matriz identidade é a matriz quadrada onde os elementos da diagonal principal são 1 e os demais 0 ($I[i,j] = 1$ se $i = j$ e $I[i,j] = 0$ se $i \neq j$). Escreva um programa em *Pascal* que leia duas matrizes e testa se a segunda é a inversa da primeira.

20. Considere uma matriz M de tamanho $N \times M$ utilizada para representar gotas de água (caractere G) em uma janela. A cada unidade de tempo T, as gotas descem uma posição na matriz, até que atinjam a base da janela e desapareçam. Considere que a chuva parou no momento em que seu programa iniciou.

    Exemplo:



    Faça um programa em que:

    (a) Leia as coordenadas iniciais das gotas de água na matriz. O canto superior esquerdo da matriz (desconsiderando as bordas) possui coordenada $(1,1)$. A coordenada $(0,0)$ indica o término da leitura. Coordenadas inválidas devem ser desconsideradas.

    Exemplo de entrada para a matriz acima (em $T = 0$):

```
1 4
1 13
4 6
2 8
100 98
4 10
0 0
```

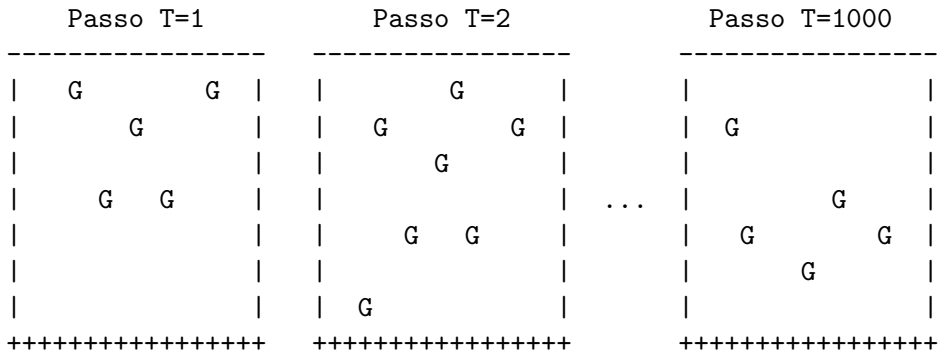
    Note que a entrada $(100, 98)$ deve ser descartada pois é inválida para a matriz do exemplo.

    (b) Imprima, a cada unidade de tempo T, o conteúdo da matriz M, atualizando a posição das gotas G até que não reste nenhuma gota na janela.

21. Modifique seu programa da questão anterior de modo que as gotas que estão inicialmente na primeira linha da janela desçam com o dobro da velocidade das outras gotas. Ou seja, as gotas que iniciam na primeira linha descem duas linhas na matriz a cada instante T. As gotas mais rápidas podem encontrar gotas mais lentas pelo caminho, neste caso a gota mais lenta desaparece ficando somente a mais rápida.

22. Modifique novamente o programa da questão anterior considerando que, desta vez, a cada unidade de tempo T, NG novas gotas são inseridas na matriz. Além disso, as gotas descem na matriz até que atinjam a base da janela e desapareçam. Inicialmente não há gotas na janela, pois a chuva começa quando $T = 1$.

    Exemplo:



    Faça um programa em que:

    (a) Leia o número de linhas (L) e o número de colunas (C) da matriz M, a quantidade de novas gotas a serem criadas a cada iteração (NG), e o número de iterações ($TMAX$) do programa.
    Exemplo de entrada para a matriz acima:

```
7 15 5 1000
```

    (b) A cada unidade de tempo T, insira NG novas gotas na matriz. A posição de uma nova gota é dada por um procedimento cujo protótipo é:

```
Procedure coordenada_nova_gota(L,C:integer; VAR x,y:integer);
```

    Este procedimento recebe quatro parâmetros: os dois primeiros indicam o número de linhas e colunas da matriz M (L, C). Os dois últimos retornam as coordenadas (x, y) da nova gota na matriz.

    (c) A cada unidade de tempo T, imprima o conteúdo da matriz M, atualizando a posição das gotas G seguindo os seguintes critérios:

        i. Quando uma gota cai sobre outra, forme-se uma gota "dupla", ou seja, ela desce duas posições a cada instante T. Caso uma nova gota caia sobre uma gota "dupla", surge uma gota "tripla", que desce três posições a cada instante T, e assim por diante.

ii. As gotas mais rápidas podem encontrar gotas mais lentas pelo caminho, neste caso a velocidade delas é somada.

23. Considere o tipo PGM para imagens como definido na seção 10.2.5. Faça um programa que leia da entrada padrão (teclado) duas imagens no formato PGM: imagem original ($imgO$) e a imagem do padrão ($imgP$).

    Em seguida, o programa deve procurar se a imagem $imgP$ está contida na imagem $imgO$ e imprimir na tela as coordenadas $(coluna, linha)$ do canto superior esquerdo de **cada ocorrência** da imagem $imgP$ encontrada na imagem $imgO$.

    Observações:

    - A imagem $imgP$ pode aparecer mais de uma vez na imagem $imgO$;
    - Na imagem $imgP$, pontos com o valor -1 devem ser ignorados, isto é, represent am pontos transparentes da imagem e não devem ser comparados com a imagem $imgO$.
    - Estruture seu código. A solução parcial ou a indicação de chamadas a funções não implementadas serão consideradas.

    Exemplo:

    - **Imagem original**:

```
P2
11 10
40
40 5  5  5  5  5  5  5  5  40 0
5  20 20 5  5  5  5  5  5  5  5
5  5  20 5  5  5  0  0  0  0  0
5  5  20 20 5  5  20 20 0  0  5
5  5  5  5  5  5  0  20 0  0  0
5  5  5  5  5  5  0  20 20 0  5
5  5  5  5  11 11 11 0  0  0  0
5  5  5  5  20 20 11 5  5  5  5
5  5  5  5  11 20 11 5  5  5  0
40 5  5  5  11 20 20 5  5  40 5
```

    - **Imagem do padrão**:

```
P2
3 3
20
20 20 -1
-1 20 -1
-1 20 20
```

    - **Resultado do Programa**:

```
2 2
7 4
5 8
```

24. Modifique o programa anterior de forma que, ao invés de imprimir as coordenadas, seja impressa uma nova imagem, que consiste de uma cópia da imagem original *imgO* na qual as ocorrências da imagem *imgP* estejam circunscritas por uma borda de um ponto de largura, com o valor máximo da imagem *imgO* (3ª linha do arquivo PGM). Você não precisa se preocupar com possíveis sobreposições das bordas.

    Exemplo da nova saída para a entrada original:

    - **Imagem resultante**:

```
P2
11 10
40
40 40 40 40 40 5  5  5  5  40 0
40 20 20 5  40 5  5  5  5  5  5
40 5  20 5  40 40 40 40 40 40 0
40 5  20 20 40 40 20 20 0  40 5
40 40 40 40 40 40 0  20 0  40 0
5  5  5  5  5  40 0  20 20 40 5
5  5  5  40 40 40 40 40 40 40 0
5  5  5  40 20 20 11 40 5  5  5
5  5  5  40 11 20 11 40 5  5  0
40 5  5  40 11 20 20 40 5  40 5
```

25. Uma matriz é chamada de *esparsa* quando possui uma grande quantidade de elementos que valem zero. Por exemplo, a matriz de ordem 5×4 seguinte é esparsa, pois contém apenas 4 elementos não nulos.

	1	2	3	4
1	0	17	0	0
2	0	0	0	0
3	13	0	-12	0
4	0	0	25	0
5	0	0	0	0

    Obviamente, a representação computacional padrão para matrizes é ineficiente em termos de memória, pois gasta-se um espaço inútil para se representar muitos elementos nulos.

    Nesta questão, vamos usar uma representação alternativa que vai permitir uma boa economia de memória.

A proposta é representar apenas os elementos não nulos. Para isto usaremos três vetores, dois deles (L e C) para guardar as coordenadas dos elementos não nulos e o terceiro (D) para guardar os valores dos elementos daquelas coordenadas. Também usaremos três variáveis para representar o número de linhas e colunas da matriz completa e o número de elementos não nulos da matriz.

Considere as seguintes definições de tipos:

```
CONST
    MAX = 6;        (* um valor bem menor que 5 x 4, dimensao da matriz *)
TYPE
    vetor_coordenadas = array [1..MAX] of integer;   (* coordenadas  *)
    vetor_elementos   = array [1..MAX] of real;      (* dados        *)
VAR
    L, C: vetor_coordenadas;  (* L: linhas, C: colunas                *)
    D: vetor_elementos;       (* D: dados                             *)
    N_lin, N_col: integer;    (* para armazenar as dimensoes da matriz *)
    N_elementos: integer      (* numero de elementos nao nulos        *)
```

Definição 1 *Um elemento M[i,j] da matriz completa pode ser obtido da representação compactada:*

- *se existe um k tal que L[k] = i e C[k] = j, então M[i,j] = D[k];*
- *caso contrário, M[i,j] = 0.*

A matriz do exemplo anterior pode então ser assim representada:

```
N_elementos:= 4; N_lin:= 5; N_col:= 4;
```

	1	2	3	4	5	6
L	1	3	3	4		
C	2	1	3	3		
D	17	13	-12	25		

(a) Fazer um procedimento que leia da entrada padrão:

- dois inteiros, representando as dimensões da matriz (linha, coluna);
- trincas de elementos l, c, d, onde l e c são inteiros e d é real, representando respectivamente a linha, a coluna e o valor de um elemento não nulo da matriz. A leitura termina quando for lido uma trinca 0, 0, 0. Para cada trinca, devem ser criados os três vetores que representam a matriz conforme descrito acima. Veja o exemplo de entrada de dados, abaixo.

Exemplo para a entrada de dados:

```
5 4
1 2 17
3 1 13
3 3 -12
4 3 25
0 0 0
```

(b) Fazer uma função que, dada uma coordenada (l, c), respectivamente para uma linha e coluna, retorne o valor de elemento M[l,c], conforme a definição 1.

(c) Fazer um procedimento que, dadas duas matrizes no formato compactado descrito acima, obtenha uma terceira matriz compactada que é a soma das duas primeiras.

(d) Fazer um procedimento que, dada uma matriz no formato compactado, imprima na tela uma matriz no formato padrão, contendo os zeros.

26. Declare uma matriz $M \times N$ de caracteres do tipo char. Implemente quatro funções que, dados como parâmetros a matriz, uma palavra do tipo string e um par de coordenadas na matriz, isto é, dois inteiros representando uma linha e uma coluna, descubram se, na matriz, a palavra ocorre iniciando na posição indicada pelas coordenadas. A primeira função procura na horizontal, da esquerda para direita; a segunda função procura na horizontal, da direita para esquerda; a terceira função procura na vertical, da cima para baixo; a quarta função procura na vertical, da baixo para cima.

27. Os incas construiam pirâmides de base quadrada em que a única forma de se atingir o topo era seguir em espiral pela borda, que acabava formando uma escada em espiral. Escreva um programa que leia do teclado uma matriz quadrada $N \times N$ de números inteiros e verifica se a matriz é inca; ou seja, se partindo do canto superior esquerdo da matriz, no sentido horário, em espiral, a posição seguinte na ordem é o inteiro consecutivo da posição anterior. Por exemplo, as matrizes abaixo são incas:

```
 1   2   3   4        1   2   3   4   5
12  13  14   5       16  17  18  19   6
11  16  15   6       15  24  25  20   7
10   9   8   7       14  23  22  21   8
                     13  12  11  10   9
```

O programa deve ler do teclado a dimensão da matriz (um inteiro N, $1 \leq N \leq 100$) e em cada uma das próximas N linhas, os inteiros correspondentes às entradas da matriz naquela linha. A saída do programa deve ser "A matriz eh inca" ou "A matriz nao eh inca".

28. Escreva um programa em *Pascal* que leia do teclado uma matriz A ($N \times M$) de inteiros e imprima uma segunda matriz B de mesma dimensões em que cada elemento $B[i, j]$ seja constituído pela soma de todos os 8 elementos vizinhos do elemento $A[i, j]$, excetuando-se o próprio $A[i, j]$.

29. Nesta questão você terá que providenciar ligações par-a-par entre diversos pontos distribuídos ao longo de uma rota qualquer. A entrada de dados consiste de um conjunto de pares $(x, y), 1 \leq x, y \leq MAX$, sendo que o último par a ser lido é o (0,0), que não deve ser processado, apenas indicando final da entrada de dados.

    Para cada par (x, y) dado como entrada, você deve providenciar uma conexão física entre eles. As linhas de uma matriz podem representar a "altura" das linhas de conexão, enquanto que as colunas da matriz podem representar os pontos (x, y) sendo conectados. Um símbolo de "`+`" pode ser usado para se representar alteração na direção de uma conexão. O símbolo "`|`" pode ser usado para representar um trecho de conexão na vertical. Finalmente o símbolo "`-`" pode ser usado para se representar um trecho de conexão na direção horizontal. Quando um cruzamento de linhas for inevitável, deve-se usar o símbolo "`x`" para representá-lo. Considere que não existem trechos de conexões na diagonal.

    Por exemplo, suponha que a entrada é dada pelos seguintes pares:

```
3 5
2 9
0 0
```

    Uma possível saída para seu programa seria a impressão da seguinte matriz:

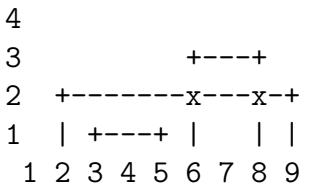
```
4
3
2  +-------------+
1  | +---+       |
 1 2 3 4 5 6 7 8 9
```

    Outra possível matriz solução para este problema seria esta:

```
4
3
2    +---+
1  +-x---x-------+
 1 2 3 4 5 6 7 8 9
```

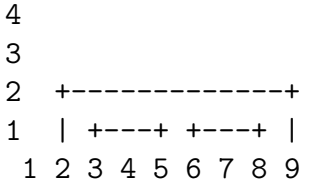
    Note que nesta última versão foi preciso inserir dois cruzamentos.

    Ainda como exemplo, se o par (6,8) também fosse dado como entrada no exemplo anterior, a saída do programa poderia ser assim exibida:



Você deve implementar um programa em *Pascal* que seja capaz de ler uma sequência de pares terminada em $(0, 0)$ (como no exemplo acima) e que imprima o desenho das conexões como saída, também conforme o diagrama acima.

30. Modifique o programa anterior com o objetivo de minizar o número de cruzamentos da matriz gerada como solução do problema anterior. Assim, a matriz ideal para ser dada como resposta do último exemplo seria a seguinte:



31. Considere o seguinte programa:

```
program prova_final;

CONST MAX=100;
TYPE matriz = array [1..MAX,1..MAX] of integer;
VAR n_lin, n_col: integer; (* dimensoes da matriz *)
    m: matriz;             (* matriz *)

(* espaco reservado para os procedimentos *)

begin
   read (n_lin, n_col);
   le_matriz (m, n_lin, n_col);
   acha_maior_sequencia (m, n_lin, n_col, l_ini, c_ini, l_fim, c_fim);
   writeln ('A maior sequencia de numeros repetidos na matriz ');
   writeln ('inicia na coordenada ', l_ini, c_ini);
   writeln (' e termina na coordenada ', l_fim, c_fim);
end.
```

Implemente os procedimentos indicados para que o programa leia uma matriz de inteiros e imprima as coordenadas de início e término da maior sequência de números repetidos da matriz. Esta sequência pode estar tanto nas linhas quanto nas colunas. No caso de existir mais de uma sequência repetida de mesmo tamanho, você pode imprimir as coordenadas de qualquer uma delas, desde que imprima as de uma só.

```
Exemplo 1:
Entrada:      Saída
4 3           1 2
1 2 3         3 2
2 2 1
3 2 5
```

```
Exemplo 2:
Entrada:      Saída
4 5           2 2
1 2 3 1 2     2 4
1 2 2 2 3
2 3 4 5 6
8 7 6 4 2
```

32. Faça um programa para:

    - ler uma sequência de polinômios $P_i(x) = a_{i_0} + a_{i_1}x + a_{i_2}x^2 + ... + a_{i_n}x^n, i = 1, 2, ..., k$;
    - A leitura deve considerar que cada linha de entrada contém um polinômio P_i. A primeira informação é o seu respectivo grau (n_i). As outras informações são os n_i coeficientes $(a_{i_0}, a_{i_1}, ..., a_{i_n})$;

      Exemplo:

      $P(x) = 8.1 - 3.4x + x^2 \Longrightarrow$ `2  8.1  -3.4  1.0`
    - A sequência de polinômios se encerra quando for fornecido um polinômio de grau zero;
    - Após a leitura de todos os polinômios, o programa deve ler uma sequência de números reais x. Para cada número real lido, o programa deve imprimir o resultado de $P_i(x)$, para todos os polinômios lidos anteriormente $(i = 1, 2, ..., k)$;
    - A sequência de números reais se encerra quando for lido o número 0.0, para o qual não se deve calcular os valores de $P_i(x)$.

    Exemplo:

```
Entrada:
2  -1.0 0.0 1.0
3  1.0  2.0  0.0  -1.0
0
4.5
1.0
0.0

Saída:
P_1(2.0) = 3.0
P_2(2.0) = -3.0
P_1(1.0) = 0.0
P_2(1.0) = 2.0
```

33. Faça um programa para:

    - ler um inteiro N e uma matriz quadrada de ordem N contendo apenas 0's e 1's.

- encontrar a maior submatriz quadrada da matriz de entrada que contém apenas 1's.
- imprimir as coordenadas dos cantos superior esquerdo e inferior direito da submatriz encontrada no item anterior. Havendo mais de uma submatriz máxima, imprimir as coordenadas de qualquer uma delas.

Exemplo: Considere a seguinte matriz quadrada de ordem 6:

	1	2	3	4	5	6
1	0	1	0	1	1	1
2	0	1	1	1	1	0
3	0	1	1	1	0	1
4	1	1	1	1	0	1
5	0	0	1	0	1	0
6	0	1	0	1	0	1

A título de ilustração, esta matriz tem:

- 22 submatrizes quadradas de ordem 1 que contém apenas 1's;
- 5 submatrizes quadradas de ordem 2 que contém apenas 1's. Por exemplo, para duas delas: uma é dada pelas coordenadas (1,4) e (2,5) e outra pelas coordenadas (2,2) e (3,3);
- 1 submatriz quadrada de ordem 3 que contém apenas 1's, as coordenadas são (2,2) e (4,4).

Como a maior submatriz quadrada que contém apenas 1's é a de ordem 3, então a saída do programa deve imprimir, para este exemplo, as coordenadas (2,2) e (4,4).

34. Escreva um programa que, dado um tabuleiro e uma lista de sub-partes retangulares do tabuleiro, retorna o número de posições que não pertencem a nenhuma sub-parte. Quando uma posição não pertence a nenhuma sub-parte dizemos que ela está *perdida*.

Entrada

A entrada consiste de uma série de conjuntos de teste.

Um conjunto de teste começa com uma linha com três números W, H e N, indicando, respectivamente, a largura e a altura do tabuleiro e o número de sub-partes deste. Estes valores satisfazem as seguintes restrições: $1 \leq W$, $H \leq 500$ e $0 \leq N \leq 99$.

Seguem N linhas, compostas de quatro inteiros X_1, Y_1, X_2 e Y_2, tais que (X_1, Y_1) e (X_2, Y_2) são as posições de dois cantos opostos de uma sub-parte. Estes valores satisfazem as seguintes restrições: $1 \leq X_1, X_2 \leq W$ e $1 \leq Y_1, Y_2 \leq H$.

O fim da entrada acontece quando $W = H = N = 0$. Esta última entrada não deve ser considerada como um conjunto de teste.

Saída

O programa deve imprimir um resultado por linha, seguindo o formato descrito no exemplo de saída.

Exemplo

```
Entrada:
1 1 1
1 1 1 1                {fim do primeiro conjunto de testes}
2 2 2
1 1 1 2
1 1 2 1                {fim do segundo conjunto de testes }
493 182 3
349 148 363 146
241 123 443 147
303 124 293 17         {fim do terceiro conjunto de testes}
0 0 0                  {fim do conjunto de testes}

Saída
Não há posições perdidas.
Existe uma posição perdida.
Existem 83470 posições perdidas.
```

10.3 Registros

Até agora vimos, como estruturas de dados, apenas vetores e matrizes. Estas estruturas são ditas *homogêneas*, no sentido que as diversas posições de memória alocadas são sempre do mesmo tipo.

Para completarmos nosso estudo básico de algoritmos, resta ainda introduzir a noção de *registros*, que são estruturas *heterogênas*, isto é, pode-se alocar várias posições de memória cada uma delas de um tipo potencialmente diferente.

10.3.1 Introdução aos registros

Assim como em um vetor ou em uma matriz, pode-se acessar diversas posições de memória com um único nome da variável, o que é diferente é que com um registro podemos misturar diversos tipos diferentes.

Por exemplo, suponhamos que seja necessário implementar um cadastro de um cliente em um banco. Normalmente, este cadastro contém: nome do cliente, telefone, endereço, sexo, idade, RG e CPF. Usando-se um registro, podemos agrupar todos os dados diferentes em uma só variável. Por exemplo, em *Pascal* podemos declarar tal variável assim:

```
var r: record
            nome: string[50];
            fone: longint;
            endereco: string;
            sexo: char;
            idade, rg, cpf: integer;
       end;
```

Cada linguagem de programação tem sua sintaxe própria para a declaração e acesso aos dados. Nos vetores e matrizes, o acesso é feito usando-se o nome da variável e um índice (ou um par no caso das matrizes). Para os registros, em *Pascal*, usa-se o nome da variável, um ponto, e o nome do campo, que é escolhido pelo programador.

Por exemplo, é válido em *Pascal* a seguinte sequência de comandos:

```
r.nome:= 'Fulano de Tal';
r.fone:= 32145678;
r.endereco:= 'Rua dos bobos, no 0';
r.sexo:= 'M';
r.idade:= 75;
r.rg:= 92346539;
r.cpf:= 11122233344;
```

Também seria válido ler a partir do teclado da seguinte maneira:

```
read (r.nome);
read (r.fone);
read (r.endereco);
read (r.sexo);
read (r.idade);
read (r.rg);
read (r.cpf);
```

Contudo, assim como se dá para o tipo *array*, para se passar um parâmetro de procedimento ou função em *Pascal* é necessário antes a declaração de um novo tipo, que poderia ser assim:

```
type cliente = record
               nome: string[50];
               fone: longint;
               endereco: string;
               sexo: char;
               idade: integer;
               rg: longint;
               cpf: qword;
           end;

var r: cliente;
```

Na verdade a linguagem *Pascal* permite uma facilidade para se economizar alguma digitação através do comando *with*. A figura 10.65 ilustra uma forma de se imprimir todo o conteúdo de um registro usando-se um procedimento. O comando *with* pode ser usado para leitura ou atribuição também.

```
procedure imprime_reg (r: cliente);
begin
      with r do
      begin
            writeln (nome);
            writeln (fone);
            writeln (endereco);
            writeln (sexo);
            writeln (idade);
            writeln (rg);
            writeln (cpf);
      end;
end;
```

Figura 10.65: Imprimindo registros.

10.3.2 Vetores de registros

O leitor pode ter observado ou pensado: um registro não faz um arquivão...

De fato, normalmente é mais comum ver os registros integrados a outras estrututas, tais como vetores, matrizes ou arquivos em disco[11]

Considerando ainda o exemplo do cliente do banco, é natural imaginar que o banco tem muitos clientes. Claro, se tiver um só, melhor fechar as portas. Uma maneira ainda um pouco precária de se manipular muitos clientes é usando a estrutura de vetores em combinação com a de registros.

A título de exemplo, consideremos então as seguintes definições:

[11] O tipo *file* está fora do escopo desta disciplina no primeiro semestre de 2009.

```
const MAX= 10000;
type cliente = record
               nome: string[50];
               fone: longint;
               endereco: string;
               sexo: char;
               idade: integer;
               rg: longint;
               cpf: qword;
         end;

      bd = array [1..MAX] of cliente;

var r: cliente;
    v: bd;
    tam_v: integer;
```

Isto é, temos um vetor de *tam_v* clientes!

Vamos imaginar que o banco é novo na praça e que é preciso criar o banco de dados contendo os clientes. Podemos usar o procedimento que é mostrado na figura 10.66.

```
procedure ler_cliente (var r: cliente);
begin
      with r do
      begin
            readln (nome);
            readln (fone);
            readln (endereco);
            readln (sexo);
            readln (idade);
            readln (rg);
            readln (cpf);
      end;
end;

procedure carregar_todos_clientes (var v: bd; var tam_v: integer);
begin
      readln (tam_v);
      for i:= 1 to tam_v do
            ler_cliente (v[i]);
end;
```

Figura 10.66: Lendo os clientes do banco.

Os algoritmos para busca, ordenação e outros tipos de manipulação desta nova estrutura devem levar em conta agora qual é o campo do registro que deve ser utilizado. Por exemplo, se quisermos imprimir o telefone do cliente do banco cujo CPF seja 1234567899, então é no campo r.cpf que devemos centrar atenção durante a busca, mas na hora de imprimir, deve-se exibir o campo r.fone. Vejamos um exemplo na figura 10.67.

```
procedure busca_telefone (var v: bd; tam_v: integer; nome_procurado: string);
var i: integer;
begin
      i:= 1;
      while (i <= tam_v) and (v[i].nome <> nome_procurado) do
            i:= i + 1;
      if i <= tam_v then
            writeln ('O telefone do cliente ',v[i].nome,' eh o ',v[i].fone)
      else
            writeln ('Cliente nao localizado na base.');
end;
```

Figura 10.67: Imprime o telefone do cliente que tem um certo CPF.

O campo do registro de interesse para um algoritmo normalmente é denominado *chave*. Por exemplo, vamos tentar ordenar o banco de dados. Por qual chave devemos fazer isto, já que temos uma estrutura contendo 7 campos diferentes? Vamos convencionar que a ordenação se dará pelo CPF do cliente. O algoritmo de ordenação pode ser, por exemplo, o método da seleção. Mas deve-se observar que, durante as trocas, *todo o registro deve ser trocado*, sob pena de misturarmos os dados dos clientes! A figura 10.68 ilustra tal situação.

O ideal é que se implemente um procedimento para as trocas, assim o código fica encapsulado e é mais fácil dar manutenção. Observar também que se a chave da ordenação for outra, por exemplo, o nome do cliente, deve-se implementar outra procedure, mudando-se o código na linha da comparação de *v[i].cpf* para *v[i].nome*.

10.3.3 Registros com vetores

Na sessão anterior vimos como implementar um vetor cujos campos são registros. Nesta sessão vamos ver como implementar um registro que tem, com um dos campos, um vetor. Para isto, vamos considerar as seguintes definições:

```
const MAX= 10000;
type vetor = array [1..MAX] of real;
     tipo_vetor = record
                    tam: integer;
                    dados: vetor;
     end;

var v: tipo_vetor;
```

A ideia é encapsular o tamanho do vetor junto com o próprio vetor. Isto facilita na hora de passar parâmetros, entre outras coisas. Em uma figura de linguagem, é como se o vetor "soubesse" seu tamanho, sem precisar passar um parâmetro indicando isto.

O conceito é simples, vamos ver pode ser feita a leitura de um vetor nesta nova estrutuda de dados. Isto é apresentado na figura 10.69.

É importante observar o correto uso dos símbolos de ponto (.) e dos colchetes, eles têm que estar no lugar certo. Uma vez que, no exemplo acima, v é uma variável

do tipo registro, ela deve receber inicialmente um ponto para se poder acessar um dos dois campos. Se quisermos acessar o tamanho, então a construção é *v.tam*. Se quisermos acessar o vetor de reais, então a construção correta é *v.dados*. Ocorre que *v.dados* é um vetor, logo, deve-se indexar com algum inteiro, por isto a construção final correta é *v.dados[i]*.

10.3.4 Observações importantes

O estudante é encorajado a praticar vários exercícios até compreender bem estas noções. Uma vez compreendido, não haverá dificuldades em prosseguir no estudo de algoritmos e estruturas de dados. A maior parte das estruturas sofisticadas que serão estudadas ao longo dos próximos anos de estudo são variantes das construções estudadas nesta seção.

Nas últimas aulas desta disciplina abordaremos problemas que podem ser resolvidos de forma elegante usando-se uma combinação de vetores, registros, matrizes e também, obviamente, os tipos básicos. Afinal, como estamos querendo mostrar desde o início do nosso estudo, a arte de se programar um computador é simplesmente encontrar as melhores estruturas de dados que, bem exploradas pelos algoritmos corretos, levam a programas eficientes e elegantes.

```
procedure ordena_por_cpf (var v: bd; n: integer);
var i, j, pos_menor: integer;
    aux: cliente;

begin
     for i:= 1 to n-1 do
     begin
          (* acha o menor elemento a partir de i *)
          pos_menor:= i;
          for j:= i+1 to n do
               if v[j].cpf < v[pos_menor].cpf then
                    pos_menor:= j;
          (* troca *)
          with aux do
          begin
               nome:= v[pos_menos].nome;
               fone:= v[pos_menor].fone;
               endereco:= v[pos_menor].endereco;
               sexo:= v[pos_menor].sexo;
               idade:= v[pos_menor].idade;
               rg:= v[pos_menor].rg;
               cpf:= v[pos_menor].cpf;
          end;

          with v[pos_menor] do
          begin
               nome:= v[i].nome;
               fone:= v[i].fone;
               endereco:= v[i].endereco;
               sexo:= v[i].sexo;
               idade:= v[i].idade;
               rg:= v[i].rg;
               cpf:= v[i].cpf;
          end;

          with v[i] do
          begin
               nome:= aux.nome;
               fone:= aux.nome;
               endereco:= aux.nome;
               sexo:= aux.nome;
               idade:= aux.nome;
               rg:= aux.nome;
               cpf:= aux.cpf;
          end;
```

Figura 10.68: Ordena pelo CPF.

```
procedure ler vetor (var v: tipo_vetor);
var i: integer;

begin
      readln (v.tam);
      for i:= 1 to v.tam do
            readln (v.dados[i])
end;
```

Figura 10.69: Lendo vetores implementados em registros.

10.3.5 Exercícios

1. Latitude e longitude são especificados em graus, (^{o}), minutos ('), segundos ("), e direção (N, S, L, O). Por exemplo, a cidade A fica na latitude 22^{o}17'34"N e longitude 53^{o}41'9"O.

   Defina um tipo em *Pascal* cujo nome seja **localizacao**, e que seja constituido de *longitude* e *latitude*, ambas do tipo **coordenadas**. Para isto voce terá que definir o tipo **coordenadas** como sendo constituido de *grau*, *minutos*, *segundos* e *direcao*.

2. Declare um vetor onde cada elemento é um registro com os campos: nome, DDD, telefone, CPF, Idade.

3. Considerando o vetor não ordenado, encontrar e imprimir o nome do cliente mais jovem. Faça um procedimento para isto

4. Ordenar por ordem de nome. Faça um procedimento para isto.

5. Dado um CPF, localizar se o nome está no vetor e imprimir todos os dados. Faça um procedimento para isto.

6. Faça um procedimento que receba por valor parâmetros para nome, DDD, telefone, CPF, Idade e o insira no vetor (que já está ordenado) em ordem alfabética. Não vale usar um vetor auxiliar. Retornar por referência o vetor alterado

7. Considere o arquivo de uma empresa (chamado de "func.dat" – um arquivo de registros) contendo para cada funcionário seu número, seu nível salarial e seu departamento. Como a administração desta empresa é feita a nível departamental é importante que no arquivo os funcionários de cada um dos departamentos estejam relacionados entre si e ordenados sequencialmente pelo seu número. Como são frequentes as mudanças interdepartamentais no quadro de funcionários, não é conveniente reestruturar o arquivo a cada uma destas mudanças. Desta maneira, o arquivo poderia ser organizado da seguinte forma:

   ```
   linha numFunc nivel departamento proximo
   0     123     7     1            5
   1     8765    12    1            -1
   2     9210    4     2            -1
   3     2628    4     3            6
   4     5571    8     2            -1
   5     652     1     1            9
   6     7943    1     3            -1
   7     671     5     3            12
   8     1956    11    2            11
   9     1398    6     1            10
   10    3356    3     1            1
   11    4050    2     2            4
   12    2468    9     3            3
   ```

Em um segundo arquivo (chamado "depto.dat" – um arquivo de registros) temos as seguintes informações:

```
codDepto nomeDepto      inicio
1        vendas         0
2        contabilidade  8
3        estoque        7
4        entrega        2
```

Assim, o primeiro funcionário do departamento de vendas é o registro 0 do arquivo de funcionários e os demais funcionários do mesmo departamento são obtidos seguindo o campo `proximo`. Ou seja, os funcionários do departamento de vendas são os funcionários nos registros: 0, 5, 9, 10 e 1. Os funcionários do departamento de contabilidade são os funcionários nos registros: 8, 11 e 4.

Escreva um programa em *Pascal* que realize as seguintes operações:

- admissão de novo funcionário
- demissão de funcionário
- mudança de departamento por um funcionário

Para estas operações devem ser lidas as informações:

- código do tipo da operação: 0 para fim, 1 para admissão, 2 para demissão e 3 para mudança de departamento
- número do funcionário
- nível salarial (somente no caso de admissão)
- número do departamento ao qual o funcionário passa a pertencer (no caso de admissão e mudança)
- número do departamento do qual o funcionário foi desligado (no caso de demissão e mudança)

O programa deve escrever as seguintes informações:

- os valores iniciais lidos dos arquivos
- para cada operação: o tipo da operação realizada, os dados da operação e a forma final dos dados (de funcionários e departamentos)

No final do programa novos arquivos "func.dat" e "depto.dat" são gerados com os dados atualizados.

Detalhamento:

- a quantidade máxima de funcionários é 1000
- a quantidade máxima de departamentos é 20

- se a quantidade máxima for ultrapassada o programa deve dar uma mensagem de erro
- se for requisitada a remoção ou mudança de um funcionário não existente no departamento especificado o programa deve dar uma mensagem de erro.
- quando for requisitada a inserção de um novo funcionário é preciso verificar se um funcionário com o mesmo número já existe.
- se o código de operação for inválido o programa deve continuar lendo um novo código até que ele seja 0 (zero), 1 (um), 2 (dois) ou 3 (três).

8. Uma empreiteira tem contratos para construir diversos tipos de casa: moderno, fazenda, colonial, etc. A quantidade de material empregada para cada tipo de casa está armazenada em um arquivo chamado "material.dat". Este é um *arquivo de registros* contendo: o tipo de casa e as respectivas quantidades de cada material necessárias para sua construção. Existem no máximo 50 tipos de casas e 100 tipos distintos de materiais.

   Cada tipo de casa é representado por um código inteiro no intervalo [1,50]. Por exemplo:

   - 1: moderno
   - 2: fazenda
   - 3: colonial, **...**

   Cada tipo de material é representado por um código inteiro no intervalo [1,100]. Por exemplo:

   - 1: ferro
   - 2: madeira
   - 3: vidro
   - 4: tinta
   - 5: tijolo, **...**

   Em um segundo arquivo, chamado "preco.dat", encontram-se os preços de cada um dos materiais. Este também é um *arquivo de registros* contendo: o código do material e o seu respectivo preço. O código do material segue a mesma codificação utilizada pelo arquivo de materiais.

   Escreva um programa *Pascal* que leia os dados do arquivo "material.dat" em uma matriz e o dados do arquivo "preco.dat" em um vetor, como ilustrado abaixo.

9. Usando as estruturas de dados abaixo escreva um procedimento em *Pascal* que recebe como parâmetro uma estrutura do tipo `TAGENDA` e ordena de forma crescente o vetor `pessoa` dessa estrutura tomando como referência para a ordenação o campo `nome` da estrutura `TPESSOA`. Ou seja, ordena uma agenda pessoal de

telefones e endereços em ordem crescente do nome das pessoas presentes na agenda. Você *deve* usar a função *compara*(r, s), que recebe dois parâmetros do tipo `string` e retorna 0 se r e s são iguais, 1 se r é lexicograficamente maior que s e -1 se r é lexicograficamente menor que s. Um nome n_1 é lexicograficamente maior que um nome n_2 se n_1 aparece depois de n_2 numa ordenação alfabética crescente desses nomes.

```
Const
  MAX = 1000;
Type
   TPESSOA = record
                nome: string;
                telefone: string;
                endereco: string
             end;
   TAGENDA = record
                pessoa: array [1..MAX] of TPESSOA;
                tamanho: integer
             end;
```

10. Uma matriz é dita esparsa quando a maioria dos seus elementos possui valor 0.0 (zero). Neste caso, a representação da matriz sob a forma tradicional (um *array* bidimensional) implica em uma utilização ineficiente da memória. Por isso, matrizes esparsas são freqüentemente representadas como vetores de elementos não nulos, sendo que cada elemento contém suas coordenadas e seu valor.

    Exemplo:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 1.2 \\ 7.3 & 0 & 99 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 17 & 0 & 0 \end{bmatrix} \Longleftrightarrow \mathbf{M_e} = \left[\begin{array}{|c|c|} \hline 1 & 4 \\ \hline 1.2 \\ \hline \end{array} \; \begin{array}{|c|c|} \hline 2 & 1 \\ \hline 7.3 \\ \hline \end{array} \; \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 99 \\ \hline \end{array} \; \begin{array}{|c|c|} \hline 3 & 2 \\ \hline 2 \\ \hline \end{array} \; \begin{array}{|c|c|} \hline 4 & 2 \\ \hline 17 \\ \hline \end{array} \right]$$

    Para representar estas matrizes em *Pascal*, podemos definir as seguintes estruturas de dados:

```
Const MAX = 1000; MAXESP = MAX*MAX/10;
Type t_matriz = record
        lin, col : integer;
        dados : array [1..MAX, 1..MAX] of real;
     end;
     elemento = record
        l, c : integer;
        val : real;
     end;
     t_matrizesp = record
        tam : integer;
        dados : array [1..MAXESP] of elemento;
     end;
```

Utilizando as estruturas de dados definidas acima, faça:

(a) Escreva uma função que transforme uma matriz do tipo `t_matriz` em uma matriz do tipo `t_matrizesp`.

(b) Escreva uma função que transforme uma matriz do tipo `t_matrizesp` em uma matriz do tipo `t_matriz`.

(c) Escreva uma função que receba duas matrizes do tipo `t_matrizesp` e imprima o resultado da **soma** destas matrizes. O resultado deve ser impresso na forma bidimensional, com os valores de cada linha separados por espaços.

11. Verdadeiro ou falso:

- um record deve ter pelo menos dois campos
- os campos de um record tem que ter nomes diferentes
- um record deve ter um nome diferente de qualquer um dos seus campos.

12. Suponha que a linguagem *Pascal* foi modificada para permitir que o símbolo ponto "."possa fazer parte de identificadores. Qual problema isto poderia causar? Dê um exemplo.

13. Esta definição é legal? Porque não?

```
TYPE
     Ponto = RECORD
                 quantidade: integer;
                 corte: Tamanho;
             END;
     Tamanho = (mini, medio, maxi);
```

14. Latitudes e longitudes são especificadas em graus (o), minutos ('), segundos (") e direção (norte, sul, leste, oeste). Suponha que uma cidade esteja na latitude 22o17'34" norte e longitude 53o41'9"oeste. Armazene esta localização na variável Cidade, como declarada abaixo:

```
TYPE
     TipoDirecao = (norte, sul, leste, oeste);
     Coordenadas = RECORD
                       graus: 0..180;
                       minutos, segundos: 0..60;
                       direcao: TipoDirecao;
                   END;
     Localizacao = RECORD
                       latitude, longitude: Coordenadas;
                   END
VAR
     Cidade: Localizacao;
```

15. Suponha que temos as seguintes definições e declarações abaixo:

```
TYPE
     NumTelefone = RECORD
                        Area, Prefixo, Numero: integer;
                   END;
VAR
     Casa, Escritorio, Celular: NumTelefone;
```

    Escreva código *Pascal* par testar se Escritorio e Celular estão sobre o mesmo código de área e para atribuir o mesmo número do Celular como sendo o do Escritório.

16. Criar registros para as seguintes configurações da vida real:

    - Pessoas com nome, endereço, telefone, sexo, idade e salário;
    - Planetas com massa, distância média do sol, número de satélites, nome, diâmetro.
    - Cursos, com alunos e suas notas, períodos em que são oferecidos, professor, horário.

17. Uma matriz é dita *esparsa* se o número de elementos não nulos for bastante pequeno com relação ao número total de elementos da matriz. Você deve fazer um programa que leia do teclado uma matriz qualquer de números *reais* e crie um vetor que armazene somente os elementos não nulos da matriz. Cada posição do vetor deve ser um registro contendo o valor do elemento, e a posição desse elemento na matriz (linha e coluna). Imprima o resultado. A maneira de definir o vetor faz parte da prova. Exemplo: Seja a matriz 4×5 abaixo:

0.0	0.0	3.0	0.0	1.7
-1.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	-2.1	0.0
0.0	0.0	2.5	0.0	0.0

    Seu programa deverá construir um vetor que permita imprimrir as seguintes informações:

    - o elemento 3.0 está na posição (1,3) do vetor;
    - o elemento 1.7 está na posição (1,5) do vetor;
    - o elemento -1.1 está na posição (2,1) do vetor;
    - o elemento -2.1 está na posição (3,4) do vetor;
    - o elemento 2.5 está na posição (4,3) do vetor;

18. Considere uma estrutura de vetores para representar uma matriz esparsa, tal como você definiu na questão anterior. Faça um procedimento para imprimir a matriz completa, contendo todos os elementos nulos e não nulos (não precisa fazer o programa todo, basta o procedimento). Por exemplo, dado vetor com as informações descritas no exemplo da questão anterior, a saída do seu programa deve ser exatamente a matriz apresentada no início do exemplo.

19. Considere o tipo PGM para imagens como definido na seção 10.2.5. Nas questões que seguem, considere as seguintes estruturas de dados e assinaturas de funções e procedimentos:

```
const MAX=10000;

type
      matriz = array [1..MAX,1..MAX] of integer;

      vetor  = array [1..MAX] of integer;

      imagem = record
                  col, lin, maior: integer;
                  m: matriz;
      end;

      imgcompactada = record
                  tam: integer;
                  v: vetor;
      end;

function calcula_valor_medio (var I: imagem): integer;
(* funcao que retorna o valor medio dos pixels da imagem, isto eh
   a soma de todos os elementos dividido pelo numero de elementos *)

procedure ler (var I: imagem);
(* procedimento que le uma imagem no formato PGM *)

procedure imprime_imagem (var I: imagem);
(* procedimento que imprime uma imagem no formato PGM *)
procedure binariza (var I: imagem; limiar: integer);
(* procedimento que transforma a imagem de tons de cinza para preto e branco
   para isto, os pixels que forem maiores que o limiar devem se tornar brancos
   e os que forem menores ou iguais a este mesmo limiar devem se tornar pretos *)

procedure compacta_imagem (var I: imagem; var C: imgcompactada);
(* procedimento que recebe uma imagem no formato PGM e cria um vetor C
   que eh uma representacao compactada desta                          *)

procedure imprime_img_compactada (var C: imgcompactada);
(* procedure que recebe uma imagem compactada e a imprime no formato PGM *)
```

   (a) Implemente estas funções e procedimentos e faça um programa que receba um certo número N de imagens PGM em tons de cinza (onde 0 representa preto e branco é representado pelo maior valor da imagem) e imprima a

imagem binarizada, isto é, em preto e branco (onde 0 representa preto e 1 representa branco). Note que o limiar é obtido pelo valor médio dos pixels.

(b) Implemente um procedimento que gere um vetor que representa a matriz binarizada de forma compacta. Para isto, use a seguinte ideia: como a matriz só tem zeros e uns, vamos substituir sequências de uns pelo número de uns consecutivos. Os elementos vão sendo colocados no vetor, de maneira linear, cada linha seguinte é concatenada à anterior. Veja o exemplo: Exemplo:

- **Imagem binarizada**:

```
P2
11 10
1
1  1  1  1  0  1  1  1  1  1  0
1  1  0  1  1  1  1  1  1  1  1
0  0  1  0  0  0  1  1  1  0  0
1  1  1  1  1  1  1  1  1  1  1
0  0  0  1  1  1  1  1  0  0  0
0  0  0  1  1  1  1  1  1  1  1
1  1  1  0  1  1  1  1  1  1  1
1  1  1  1  1  1  1  0  1  1  1
1  1  1  1  1  1  1  1  1  0  0
0  1  1  1  1  1  1  1  1  1  1
```

- **Imagem compactada**:

```
36
4 0 5 0 2 0 8 0 0 1 0 0 0 3 0 0 11 0 0 0 5 0 0 0 0 0 0 11 0 14 0 12 0 0 0
```

Isto é, a primeira linha da matriz possui 4 uns consecutivos seguido de um zero e outros 5 uns consecutivos, por isto, o vetor contém seus primeiros elementos “4, 0 e 5”. Preste atenção antes de escrever o código. Você pode definir, se precisar, funções, procedimentos ou estruturas de dados adicionais.

Capítulo 11

Desenvolvendo programas de maior porte

Neste ponto do curso já temos todas as noções fundamentais de algoritmos e estruturas de dados que são necessárias para a construção de programas complexos. O objetivo daqui para frente é mostrar como se desenvolver programas de maior porte em relação aos que vimos até agora.

11.0.1 Campo minado

O primeiro grande problema que vamos tentar resolver é o conhecido jogo do *Campo minado*, costumeiramente disponível gratuitamente em diversos sistemas operacionais modernos. Uma descrição do funcionamento deste jogo pode ser encontrada na Wikipedia[1].

Vamos tentar implementar a versão ASCII do jogo, contudo com algumas simplificações: vamos ignorar as bandeiras que se podem colocar para marcação de prováveis bombas e também vamos ignorar a contagem do tempo e registro dos recordes. Como a interface é ASCII, naturalmente também não vamos nos preocupar com a beleza da interface.

A ideia é desenvolvermos um código final usando-se a técnica de refinamentos sucessivos, isto é, vamos partir de um código geral, muitas vezes escritos em pseudo-linguagem, e a cada etapa detalharmos as estruturas de dados e algoritmos necessários para a resolução do problema. Para isto vamos explorar ao máximo as noções de funções e procedimentos da linguagem *Pascal*.

Vamos partir da existência de um tabuleiro de dimensões $N \times M$ que contém um certo número B de bombas em algumas posições. As outras posições contém o número total de bombas vizinhas. A cada jogada, o usuário entra com as coordenadas (i, j) onde deseja abrir o tabuleiro, se for uma bomba, o jogo acaba, senão é exibido o valor da célula. O jogo também termina quando o usuário abriu todas as posições que não tem bombas.

O pseudocódigo que resolve este problema está ilustrado na figura 11.1. Neste ponto do desenvolvimento, optamos por postergar as definições das estruturas de

[1] `http://pt.wikipedia.org/wiki/Campo_minado`

dados necessárias.

```
begin (* programa principal *)

    ''Criar o campo minado de dimensoes NxM contendo B bombas''

    repeat
        ''Imprimir o tabuleiro ocultando as celulas nao abertas''
        ''Ler as coordenadas da celula a ser aberta''
        ''Abrir a celula indicada''
        ''Se nao era bomba, entao
                atualiza bombas restantes''
        ''Senao
                jogador perdeu''
    until ''jogador ganhou'' or ''jogador perdeu'';

    ''Imprimir o tabuleiro final'';
end.
```

Figura 11.1: Pseudocódigo para campo minado.

Podemos imaginar algumas funções e procedimentos contendo alguns parâmetros parcialmente definidos, mas já com o comportamento desejado. Por exemplo, vamos supor a existência dos seguites:

CriaCampoMinado (campo): Procedimento que cria um campo minado contendo dimensões $N \times M$ com B bombas de maneira que cada elemento deste campo minado, quando clicado, exibe uma bomba ou o número de bombas ao redor da posição. Inicialmente este campo é criado de maneira que nenhuma célula ainda foi aberta.

ImprimeCampoMinado (campo): Procedimento que imprime o campo minado de maneira que as células ainda não abertas não são exibidas ao usuário. Para as que já foram abertas é exibido o número de células vizinhas contendo bombas ou uma bomba.

LerCoordenadas (i, j, acao): Procedimento que lê um par de coordenadas representando respectivamente a linha i e a coluna j de uma célula para ser aberta. O procedimento garante que as coordenadas são válidas. Ação define o que fazer com a coordenada, se for zero é para abrir a célula, se for 1, é para marcar com uma bandeira e se for 2 é para desmarcar a bandeira.

AbrirCelula (campo, i, j): Função que abre a célula (i, j) do campo minado e retorna *false* caso a célula contenha uma bomba. Senão, retorna *true*. No caso da célula conter uma bomba, também retorna o campo minado com a informação de que todas as células do campo são agora consideradas abertas. No caso contrário, considera como abertas as células da seguinte maneira: se a célula contém um número diferente de zero bombas, abre somente a coordenada (i, j). Se for um valor de zero bombas, então considera como abertas todas as

células vizinhas de (i, j). Se uma destas também contiver zero bombas vizinhas, então também abre suas vizinhas, de maneira iterativa, até que não hajam mais células alcançáveis cujo número de bombas vizinhas seja zero.

Com estes procedimentos, é possível *refinar* a versão em pseudocódigo acima para a que está mostrada na figura 11.2. Notem que ainda não especificamos as estruturas de dados que permitem a implementação destes procedimentos e funções.

```
begin
    CriaCampoMinado (campo);

    repeat
        ImprimeCampoMinado (campo);
        LerCoordenadas (i, j, acao);
        if AbrirCelula (campo, i, j) then
                ''atualiza bombas restantes''
        else
                ''jogador perdeu''
    until ''jogador ganhou'' or ''jogador perdeu'';

    ImprimeCampoMinado (campo);
end.
```

Figura 11.2: Primeiro refinamento para campo minado.

Para fecharmos o código do programa principal, no próximo refinamento temos que definir como o jogo termina, isto é, como determinarmos quando o jogador perdeu e quando ele ganhou. O primeiro caso é fácil, o jogador perde quando clica em uma bomba.

Quanto ao outro caso, basta, por exemplo, determinar quantas foram as células abertas com sucesso, isto é, sem provocar explosão de bombas. O problema é que a função *AbrirCelula* abre, em potencial, mais células do que a que foi clicada. Isto ocorre quando se entra com uma coordenada de célula que contém zero bombas vizinhas. Para resolver este problema, temos que modificar o comportamento da função, prevendo que ela contém um parâmetro adicional que informa o número de células que ela abriu, incluindo a da coordenada fornecida. Assim:

AbrirCelula (campo, i, j, nCasasAbertas): Em adição à descrição acima, o parâmetro *nCasasAbertas* retorna o número de células que foram abertas durante o procedimento, incluindo a coordenada (i, j), em caso de sucesso. Este valor é zero quando uma bomba foi encontrada.

A rigor, podemos modificar a função para que ela retorne não um valor booleano, mas simplesmente retorne o número de casas que foram abertas. Se o valor for zero é porque se escolheu uma bomba.

Então o procedimento fica assim:

AbrirContarCelulasAbertas (campo, i, j): Função que abre a célula (i, j) do campo minado e retorna *zero* caso a célula contenha uma bomba. Senão, retorna o número de células abertas. No caso da célula conter uma bomba, também retorna o campo minado com a informação de que todas as células do campo são agora consideradas abertas. No caso contrário, considera como abertas as células da seguinte maneira: se a célula contém um número diferente de zero bombas, abre somente a coordenada (i, j). Se for um valor de zero bombas, então considera como abertas todas as células vizinhas de (i, j). Se uma destas também contiver zero bombas vizinhas, então também abre suas vizinhas, de maneira iterativa, até que não hajam mais células alcançáveis cujo número de bombas vizinhas seja zero.

Vamos precisar também saber o número de bombas colocadas no campo minado e as dimensões deste, o que pode vir de uma função que retorna o número de células do campo (função *nCelulas (campo)*). Assim o programa pode mais uma vez ser refinado, ficando como é mostrado na figura 11.3.

```
var CelulasParaAbrir, nBombas: integer;
    perdeu: boolean;
begin
    CriaCampoMinado (campo, nBombas);
    CelulasParaAbrir:= nCelulas (campo) - nBombas;
    perdeu:= false;

    repeat
        ImprimeCampoMinado (campo);
        LerCoordenadas (i, j, acao);
        nAbertas:= AbrirContarCelulasAbertas (campo, i, j, acao);
        if nAbertas = 0 then
            perdeu:= true;
        else
            CelulasParaAbrir:= CelulasParaAbrir - nAbertas;
    until (CelulasParaAbrir = 0) or perdeu;

    ImprimeCampoMinado (campo);

end.
```

Figura 11.3: Segundo refinamento para campo minado.

Agora vamos precisar definir as estruturas de dados para podermos implementar os procedimentos acima e garantir que o jogo funcione. Mas o estudante deve notar que existem diversas maneiras de se definir estruturas de dados para este problema. A que vamos mostrar é apenas uma dentre as possibilidades e foi escolhida pois também vai servir para treinarmos o uso dos conceitos até aqui estudados. O aprendiz é encorajado a buscar alternativas e tentar um código mais eficiente do que este que será apresentado.

Com estas observações em mente, vamos definir um campo minado como sendo uma matriz de inteiros. Cada valor representará o número de bombas ao redor de

uma célula. Para não complicarmos demais a estrutura, vamos considerar que uma célula contendo uma bomba é representado por um número negativo, por exemplo, o -1. Esta é uma boa escolha, pois o número de bombas vizinhas é sempre maior ou igual a zero e sempre menor do que nove. A rigor, poderíamos usar qualquer valor fora deste intervalos, mas vamos manter o -1.

Esta estrutura é insuficiente para se armazenar o fato das células já terem sido abertas ou não, uma vez que somente podem ser impressas as que já foram clicadas alguma vez. Por isto, vamos adotar um registro para ser o elemento da matriz: um campo informa um número inteiro e o outro informa se a célula deve ser mostrada ou não.

Finalmente, há a questão das bandeiras que marcam posições onde o jogador acredita conterem bombas. Vamos usar mais um campo booleano para representar isto. Desta maneira, nossa estrutura de dados para representar o tabuleiro é apresentada a seguir.

```
celula = record
                nbombas: integer;
                aberto: boolean;
                marca: boolean;
         end;
matriz = array [1..max,1..max] of celula;

campominado = record
                nlin,ncol: integer;
                m: matriz;
              end;
```

Assim o tabuleiro é um registro que contém uma matriz e as dimensões desta. Cada elemento da matriz é um registro que armazena o número de bombas (-1 representando uma bomba) e dois campos booleanos para se representar uma célula que já foi aberta e se tem uma bandeira ou não.

Com isto podemos escrever código para as funções e procedimentos. Vamos iniciar pelo primeiro que aparece no código, isto é, a inicialização da estrutura de dados: *CriaCampoMinado*. Este código é exibido na figura 11.4.

```
procedure CriaCampoMinado (var campo: campominado; var nbombas: integer);

begin
    readln(campo.nlin, campo.ncol); (* le as dimensoes do campo      *)
    readln(nbombas);                (* le numero de bombas no campo *)
    zera_campo (campo);             (* inicializa o campo vazio     *)
    gera_bombas (campo, nbombas);   (* coloca as bombas no campo    *)
    conta_vizinhos (campo);         (* conta os vizinhos com bombas *)
end;
```

Figura 11.4: Criando campo minado.

O leitor pode observar que optamos por ainda não definir alguns códigos de procedimentos, mas já é necessário conhecer a estrutura do campo para que se possam armazenar no lugar certo os valores do número de linhas e colunas.

Os outro procedimentos são apresentados a seguir. Zerar o campo é necessário pois haverá uma distribuição de bombas e a posterior contagem dos vizinhos que contém bombas, logo, o programador deve inicializar todas as estruturas para que o procedimento completo seja robusto. O procedimento para zerar o campo é apresentado na figura 11.5.

```
procedure zera_campo (var campo: campominado);
var i,j: integer;
begin
      for i:= 1 to campo.nlin do
            for j:= 1 to campo.ncol do
            begin
                  campo.m[i,j].nbombas:= 0;
                  campo.m[i,j].aberto:= false;
                  campo.m[i,j].marca:= false;
            end;
end;
```

Figura 11.5: Criando campo minado.

Este procedimento percorre toda a matriz e zera o número de bombas ao mesmo tempo em que informa que, inicialmente, nenhuma célula do campo pode ser exibida ainda. Também não existem bandeiras a serem mostradas.

O procedimento para gerar as bombas será definido assim: tendo-se o número de bombas que devem ser distribuídas, vamos gerar aleatoriamente as posições (linha e coluna) onde elas serão colocadas. Isto é apresentado na figura 11.6.

```
procedure gera_bombas (var campo: campominado; nbombas: integer);
var i, l, c: integer;
begin
      randomize;
      for i:= 1 to nbombas do
      begin
            l:= random (campo.nlin) + 1;
            c:= random (campo.ncol) + 1;
            campo.m[l,c].nbombas:= -1; (* -1 eh uma bomba *)
      end;
end;
```

Figura 11.6: Distribuindo as bombas.

É importante observar que este procedimento pode gerar coordenadas iguais para bombas diferentes, e isto pode ocasionar que o campo tem menos bombas do que o previsto. O estudante é encorajado a modificar este procedimento para garantir que nunca uma bomba pode ser colocada em uma posição que já contenha outra.

Neste ponto temos uma matriz zerada a menos das posições que contém bombas. Não foi alterada nenhuma informação sobre células abertas ou não.

O procedimento para contar vizinhos é, a princípio, simples. Basta varrer a matriz e para cada célula diferente de -1 se percorrer todos os oito vizinhos e contar as bombas.

O problema é que as células nas bordas da matriz vão fazer um código muito extenso, repleto de *if-then-else*'s. Por isto vamos optar por modificar a estrutura de dados, prevendo uma *borda* para o programador que facilite o procedimento. Esta borda, desde que todos os elementos estejam zerados, ajuda bastante, mas o programador só deve usá-la neste caso, isto é, no caso de contar as bombas. O código é mostrado na figura 11.7.

```
procedure conta_vizinhos (var campo: campominado);
var i, j, k, l, nbombas: integer;
begin
    for i:= 1 to campo.nlin do
        for j:= 1 to campo.ncol do
        begin
            if campo.m[i,j].nbombas <> -1 then
            begin
                nbombas:= 0;
                for k:= i-1 to i+1 do
                    for l:= j-1 to j+1 do
                        if campo.m[k,l].nbombas = -1 then
                            nbombas:= nbombas + 1;
                campo.m[i,j].nbombas:= nbombas;
            end;
        end;
end;
```

Figura 11.7: Contando vizinhos com bombas.

Notem que agora a estrutura de dados passou por uma revisão, pois deve prever a existência da borda. Agora os índices iniciam-se em zero e terminam em max+1.

```
    matriz = array [0..max+1,0..max+1] of celula;
```

O resto permanece igual, em termos de estrutura de dados, mas o algoritmo que zera o tabuleiro tem que ser revisto, pois as bordas têm que ser inicializadas também. A modificação é simples, bastando se alterar os limites dos dois comandos *for*, conforme a figura 11.8:

```
procedure zera_campo (var campo: campominado);
var i,j: integer;
begin
    for i:= 0 to campo.nlin + 1 do
        for j:= 0 to campo.ncol + 1 do
            (* o resto nao muda *)
end;
```

Figura 11.8: Criando campo minado.

Neste ponto temos um campo minado pronto para ser jogado!

Mas falta definirmos como será feita a impressão do tabuleiro, como se entram com as coordenadas e, finalmente, como se abre uma célula. Os procedimentos para

leitura das coordenadas e impressão da matriz não será exibido aqui por serem muito simples. Mas também pode o estudante que souber usar o mouse implementá-los como uma rotina gráfica. Mostramos na figura 11.9 o procedimento que abre uma célula.

```
function AbrirContarCelulasAbertas (var campo: campominado; lin, col, acao: integer)
                                        : integer;
var r: coordenadas;
    cont: integer;
    abrir: conjunto;

begin
     if acao = 0 then
          if campo.m[lin,col].nbombas = -1 then (* achou bomba... *)
               AbrirContarCelulasAbertas:= 0
          else
          begin
               cont:= 1;
               campo.m[lin,col].aberto:= true;
               campo.m[lin,col].marca:= false;
               if campo.m[lin,col].nbombas = 0 then
               begin
                    r.lin:= lin;
                    r.col:= col;
                    insere_no_conjunto (abrir, r);
               end;
               ''AbreTodosVizinhosSemBomba'';
               AbrirContarCelulasAbertas:= cont;
          end
     else
          if acao = 1 then
               campo.m[lin,col].marca:= true
          else
               if acao = 2 then
                    campo.m[lin,col].marca:= false;
end;
```

Figura 11.9: Criando campo minado.

Este procedimento é o mais complexo de todos a implementar. De fato, quando se abre uma célula que tem zero bombas vizinhas, toda sua vizinhança deve ser aberta também. O problema é que um destes vizinhos (ou mais de um) pode também contém um valor zero. Por isto usamos um procedimento que abre todos os vizinhos que têm zero bombas e os vizinhos destes e assim por diante, até não sobrar mais nenhum (*AbreTodosVizinhosSemBomba*).

A solução mais elegante para este problema envolve a noção de *recursividade* que está fora do escopo deste curso, normalmente ela é vista em Algoritmos e Estruturas de Dados II.

A solução adotada foi, para cada coordenada que possui zero bombas vizinhas, armazenar esta coordenada para ser aberta posteriormente em outro procedimento.

Por isto criamos um tipo de dados chamado *conjunto*, que contém as coordenadas da posição em questão.

A ideia é temos um procedimento que pega um elemento do conjunto, sabidamente uma célula que tem zero bombas vizinhas, e que abre todos os vizinhos, mas para os que são nulos, armazena no conjunto. Assim, enquanto o conjunto contiver elementos, existem células para serem abertas ainda. O código pode ser visto na figura 11.10.

```
procedure AbreTodosVizinhosSemBomba (var campo: campominado;
                                     var abrir: conjunto;
                                     var cont: integer);
var r, q: coordenadas; i, j: integer;

begin
     while not conjunto_vazio (abrir) do
     begin
          remove_do_conjunto (abrir,r);
          for i:= r.lin-1 to r.lin+1 do
               for j:= r.col-1 to r.col+1 do
               begin
                    if DentroCampo (campo, i, j) then
                    begin
                         if (campo.m[i,j].nbombas = 0) and
                            not campo.m[i,j].aberto then
                         begin
                              q.lin:= i;
                              q.col:= j;
                              insere_no_conjunto (abrir, q);
                         end;
                         if  not campo.m[i,j].aberto then
                         begin
                              campo.m[i,j].aberto:= true;
                              cont:= cont + 1;
                         end;
                    end;
               end;
     end;
end;
```

Figura 11.10: Criando campo minado.

. O problema agora é definirmos a estrutura de dados para este novo tipo, necessário para o funcionamento do código, chamado *conjunto*. Também criamos uma função chamada DentroCampo para informar se uma coordenada está dentro do campo minado, pois podemos estar pegando elementos da borda.

A definição do tipo *conjunto* será feito na próxima seção, mas podemos neste momento imaginar que as funções e procedimentos que manipulam os dados deste tipo funcionam corretamente e declarar que o código que resolve o problema do campo minado está concluído.

A título de exemplo, colocamos o código completo na página web da disciplina: `http://www.inf.ufpr.br/cursos/ci055/mines_v4.pas`.

11.1 Exercícios

1. Escreva um programa em *Pascal* que implemente o jogo "caça palavras". O seu program deve ler da entrada padrão:

   - o número de linhas da matriz de letras
   - o número de colunas da matriz de letras
   - a matriz de letras (seqüência de letras)
   - o número de palavras
   - a lista de palavras, sendo cada palavra dada por:
     - o tamanho da palavra
     - a palavra (seqüência de letras)

   O programa deve procurar cada palavra da lista nas linhas e nas colunas da matriz. Para cada palavra encontrada o programa deve informar as posições (linha, coluna) de inicio e fim da palavra na matriz.

2. Abaixo temos parte da implementação de um jogo de caça-palavras. Neste jogo de caça-palavras as palavras podem estar dispostas apenas na horizontal (da esquerda para direita apenas) ou na vertical (de cima para baixo). A entrada da matriz de letras deve ser feita a partir de um arquivo e as palavras devem ser entradas a partir do teclado. Para cada palavra a ser procurada devem ser impressa as coordenadas da matriz onde a palavra se inicia e onde termina. Você deve completar esta implementação para que o programa funcione corretamente.

```
Program Caca_palavras (input,output,arquivo);

(* declare as constantes necessárias *)
(* declare os tipos necessários *)
(* implemente os procedimentos e funções necessários *)
(* declare as variáveis necessárias *)

BEGIN
        ASSIGN (arquivo,'matriz.txt');
        RESET (arquivo);
        Leia (tabuleiro, numLinhas, numColunas);
        CLOSE (arquivo);
        WRITELN ('entre com uma palavra qualquer');
        READ (palavra);
        WHILE NaoVazia (palavra) DO
        BEGIN
            IF Achou (palavra,tabuleiro,linIni,colIni,linFim,colFim) THEN
            BEGIN
                 WRITE ('A palavra ',palavra,' ocorre a partir da coorden
```

```
                    WRITELN (linIni,colIni,' terminando em ',linFim,colFim);
                END;
                READ (palavra);
            END;
    END.
```

Questão 2
(60 pontos)

Considere as seguintes definições de tipos:

```
TYPE
     coordenada = RECORD
                       linha, coluna: integer;
                  END;
     vetor = ARRAY [1..MAXTAM] of coordenada;
     ocorrencias = RECORD
                       numOcorre: integer;
                       coord    : vetor;
                  END;
     tabela = ARRAY ['A'..'Z'] of ocorrencias;
```

A idéia consiste em tentar otimizar a busca de palavras armazenando as ocorrências de cada letra da matriz de entrada em uma tabela de ocorrências. Na tabela podemos registrar não apenas quantas vezes cada letra ocorre na matriz mas também as coordenadas de todas estas ocorrências.

(a) Faça um desenho da estrutura de dados "tabela" acima definida e explique como ela pode ser usada para que possamos saber, para cada letra, as coordenadas de todas as ocorrências desta letra na matriz do caça-palavras;

(b) Crie um procedimento que recebe como parâmetros a matriz do jogo, suas dimensões, e inicialize a estrura da tabela conforme você acaba de explicar no item anterior;

(c) Implemente a função "Achou" do programa anterior de maneira a tirar onde ela ocorre, que podem ser armazenadas em um vetor. proveito da estrutura "tabela" aqui definida.

3. Um passatempo bastante comum são os "caça-palavras". Neste tipo de problema existe uma lista de palavras sobre algum tema e uma grade repleta de letras distribuídas de maneira aparentemente sem sentido. Por exemplo:

   - chave
   - mel
   - erva
   - cama
   - véu

s	c	C	e	l	r
U	e	H	A	y	r
g	E	A	i	M	c
c	A	V	R	E	A
c	x	E	p	L	a

Resolver o problema consiste de encontrar na grade as palavras indicadas na lista. Neste exercício você deve projetar um programa que receba como entrada um problema do tipo "caça-palavras" e deve retornar as coordenadas de início e término de cada palavra fornecida na lista de palavras. Valerão pontos: a correta escolha da estrutura de dados, o bom uso de funções e procedimentos, a modularidade, a possibilidade de generalização do problema, por exemplo, se você tiver um dia que modificar o seu programa para um caça-palavras tridimensional, seu programa deve ser facilmente adaptável. Note que as palavras estão em qualquer direção, horizontal, vertical, nas diagonais e que podem ser escritas normalmente ou de maneira inversa. Preocupe-se inicialmente em detalhar a estrutura do programa, isto já valerá pontos. Mas observe que valerá a totalidade dos pontos apenas as questões que detalharem código de maneira suficiente.

4. Considere um dos mais populares jogos de azar do momento, a Mega-Sena. Neste jogo, um certo número de apostadores marcam em uma cartela um número variável entre 6 e 15 dezenas (números inteiros) entre 1 e 60. Após o encerramento das apostas, a agência que detém o controle do jogo (doravante denominada Agência) promove um sorteio público de exatamente seis dezenas. Estas dezenas sorteadas são comparadas com cada uma das apostas feitas pelo público. Todos as cartelas que contiverem as seis dezenas sorteadas são consideradas ganhadoras e dividem o prêmio.

   Neste exercício você deve imaginar que foi contratado pela Agência para projetar e implementar um sistema computacional que seja capaz de ler um arquivo contendo o conjunto das apostas do público e encontrar o conjunto de vencedores. Por questões de tempo, e para facilitar a correção, você deve seguir as seguintes instruções:

   - Use a primeira metade do tempo da prova para projetar o sistema, iniciando pelo programa principal. Defina, na medida do necessário, as estruturas de dados que você utilizará. Pense em usar constantes, tipos e variáveis adequadas para o algoritmo que você está planejando. Pense em deixar partes importantes do programa a cargo de procedimentos ou funções.
   - Use e abuse de procedimentos e funções (doravante apenas procedimentos). Você não é obrigado a implementar todos os procedimentos. Mas você *deve* indicar claramente seus protótipos e fazer uma documentação adequada de cada um. Lembrando que um protótipo de procedimento é a linha que contém a palavra reservada PROCEDURE (ou FUNCTION), o nome, os argumentos e o tipo de retorno (no caso das funções), isto é, é o procedimento tirando o que vem entre o BEGIN e o END (inclusive).
   - Você receberá pelo menos duas folhas de resposta. Organize-as para que as primeiras páginas contenham a declaração de tipos, constantes e variáveis, os protótipos de funções que você não vai implementar, etc. Procure reservar uma página inteira também para o programa principal, que, a princípio, não é grande.

- Você terá uma certa liberdade para estruturar o sistema. Mas lembre-se que a escolha das estruturas de dados é decisiva!

Você deverá implementar completamente o programa principal, as declarações de constantes, tipos e variáveis utilizadas e no mínimo procedimentos para fazer o seguinte:

(a) Ler o conjunto de apostas, que deverá estar armazenado em um arquivo do tipo TEXT. Este arquivo conterá uma linha para cada aposta e cada aposta consiste de um identificador da aposta e dos números apostados (entre 6 e 15 dezenas);

(b) Gerar aleatoriamente uma aposta válida;

(c) Verificar se uma dada aposta é vencedora, baseando-se evidentemente na combinação sorteada;

(d) Gerar uma coleção de seis dezenas válidas que maximizam o número de apostas ganhadoras.

5. Fazendo uso das boas técnicas de programação vistas durante o curso, faça um programa em *Pascal* que implemente um *jogo da velha*:

   - O jogo possui um tabuleiro composto de nove posições, na forma de uma matriz de tamanho 3 por 3; cada posição pode estar vazia ou pode ser ocupada pelo símbolo de um dos jogadores.
   - Dois jogadores participam do jogo, sendo que a cada um destes é associado um símbolo distinto, por exemplo: "X" e "0".
   - A primeira jogada é efetuada pelo jogador X; em cada jogada um dos jogadores ocupa uma posição vazia do tabuleiro; os jogadores se alternam a cada jogada.
   - Um dos jogadores vence quando ocupa uma posição que completa uma seqüência de três símbolos iguais em uma linha, coluna ou diagonal.
   - O jogo termina empatado quando todas as posições do tabuleiro foram ocupadas e não houve vencedor.

6. Considere o problema do "Jogo da Vida" estudado em sala de aula. Sem qualquer sombra de dúvidas, a operação mais cara computacionalmente falando em qualquer algoritmo que implemente a solução do jogo é a contagem do número de vizinhos de cada célula. Considerando uma célula fora da borda do tabuleiro, escreva um trecho de código *Pascal* que obtenha o número de vizinhos de uma célula. Você **não** pode fazer uso de **nenhum** comando *repeat*, *while*, *if*, ou *case*, mas deve usar comandos *for*. Este trecho de código deverá conter apenas e tão somente a definição do tipo de dados para o tabuleiro e os comandos para contar o número de vizinhos.

7. Na questão anterior foi solicitado para não se considerar a borda do tabuleiro. Nesta questão você deve indicar, segundo a estrutura de dados definida no item anterior, qual o comportamento do seu programa quando se tenta contar o número de vizinhos de uma célula da borda. Se por acaso houver alguma possibilidade de erros, indique como deve ser modificado o algoritmo, sem alterar a estrura de dados, para que o erro deixe de existir.

8. Considere um jogo de Batalha Naval em que cada participante disporá seus 5 barcos de 1, 2, 3, 4 e 5 células, respectivamente, no espaço de uma matriz de $N \times N (N \geq 100)$.

   Os barcos deverão estar ou na direção horizontal ou na vertical, deverão ser retos e não poderão se tocar. Não poderá haver barcos que passem pelas linhas e colunas marginais da matriz.

   Escreva o programa principal para montar um jogo e fazer uma disputa entre dois adversários, especificando as chamadas às diferentes funções e procedimentos.

   Escrever as funções e procedimentos necessários para o bom funcionamento do seu programa principal acima.

   Você deve documentar a lógica da solução de forma precisa. Em particular, descreva as estruturas de dados que você utilizar e a forma como elas serão usadas para resolver o problema.

9. Implementar o Jogo da vida.

10. implementar o Sudoku

11. implementar a TV da vovó

12. implementar a espiral.

13. implementar o minesweeper.

Capítulo 12

Tipos abstratos de dados

No capítulo anterior deixamos em aberto a definição de um tipo de dados do qual não importava saber, naquele momento, quais os detalhes das estruturas de dados e aloritmos, mas importava saber da existência de funções e procedimentos com comportamentos específicos para garantir o funcionamento do jogo do campo minado.

Esta ideia será desenvolvida aqui neste capítulo, isto é, vamos mostrar como definir *tipos abstratos de dados*. Como no exemplo do tipo conjunto, a ideia e termos interfaces bem definidas. A implementação pode ser feita de várias maneiras, mas na verdade, para quem usa, não faz diferença alguma, a não ser o *comportamento* destas funções e procedimetnos.

Vamos iniciar pelo tipo *conjunto*, apenas para concluir o exercício do capítulo anterior. Depois mostraremos outros tipos, todos compatíveis com o conteúdo desta disciplina.

12.1 Tipo Conjunto

Um conjunto é um conceito matemático bem conhecido. Trata-se de uma coleção de elementos, sem repetição. Operações importantes sobre conjuntos incluem saber se o conjunto é vazio, se um elemento pertence ou não a um conjunto, inserir ou remover elementos ou realizar operações de união e interseção.

Primeiramente, vamos definir as interfaces, deixando a implementação para um segundo momento. As principais operações sobre conjuntos são as seguintes:

CriaConjunto (C): Inicializa a estrutura de dados;

ConjuntoVazio (C): Função que rotorna *true* se o conjunto C é vazio e *false* caso contrário;

Cardinalidade (C): Função que retorna a cardinalidade (o número de elementos) do conjunto C.

Pertence (x,C): Função que retorna *true* se o elemento x pertence ao conjunto C e *false* caso contrário;

Insere (x, C): Procedimento que insere o elemento x no conjunto C garantindo que não haverá repetição;

Remove (x,C): Procedimento que remove o elemento x do conjunto C;

Uniao (C1, C2, C3): Procedimento que faz a união dos conjuntos $C1$ e $C2$ retornando o resultado no conjunto $C3$;

Interseccao (C1, C2, C3): Procedimento que faz a intersecção dos conjuntos $C1$ e $C2$ retornando o resultado no conjunto $C3$;

Contido (C1, C2): Função que retorna *true* se o conjunto $C1$ está contido no conjunto $C2$ e retorna *false* caso contrário.

Vamos para isto usar um conjunto de números reais. As operações aqui definidas servirão para dar a base necessária para que se possa implementar o conjunto de coordenadas para o exercício do campo minado, bastando para isto que se adapte o tipo real para o tipo coordenadas.

O conjunto de reais será implementado em um registro com dois campos: um vetor de reais e o tamanho útil desse.

```
const max = 10000;
type
    vetor = array [1..max] of real;

    conjunto = record
                    tam: integer;
                    v: vetor;
               end;
```

Figura 12.1: Estrutura de dados para tipo conjunto.

As funções e procedimentos terão que lidar com o tamanho máximo deste vetor, que é definido pela constante *max*. Vamos considerar a existência de um procedimento denominado *Erro (msg)* que retorna erro em caso de qualquer problema apresentando a mensagem *msg*. Evidentemente que isto pode ser feito de outra maneira, mas esta opção é apenas para facilitar a redação do texto.

Vamos implementar na ordem os procedimentos acima descritos. O primeiro deles é o que inicializa a estrutura, que deve resultar na criação de um conjunto vazio. Para isto, basta definirmos que o tamanho do vetor é zero, conforme mostrado na figura 12.2.

Assim, é fácil também definirmos o código da função que testa se o conjunto é ou não vazio, conforme mostrado na figura 12.3.

Também é trivial a função que retorna a cardinalidade de um conjunto, o que é apresentado na figura 12.4.

A próxima tarefa é criar um procedimento que define se um elemento pertence ao conjunto. Para isto, é necessário que se percorra o vetor em busca do elemento. Isto já foi feito nesta disciplina. Pode-se usar busca simples, busca com sentinela, busca

```
procedure CriaConjunto (var c: conjunto);
begin
    c.tam:= 0;
end;
```

Figura 12.2: Procedimento para criar um conjunto vazio.

```
function ConjuntoVazio (var c: conjunto): boolean;
begin
    if c.tam = 0 then ConjuntoVazio:= true
                 else ConjuntoVazio:= false;
end;
```

Figura 12.3: Função que testa se conjunto é vazio.

binária ou outra que se venha a inventar. No momento, vamos implementar uma busca simples, sem sentinela, conforme mostrado na figura 12.5.

A próxima tarefa é garantir a inserção de um elemento no conjunto, de modo a não gerar elementos repetidos. Isto é mostrado na figura 12.6. Optamos por inserir sempre no final, para tornar a operação menos custosa. Mas poderiamos ter optado por outra maneira de inserir, por exemplo, sempre no início, de maneira ordenada (por exemplo para viabilizar busca binária) ou outras. Isto é importante para se implementar outras funções ou procedimentos como os de remoção, união, etc. . . .

Para remover um elemento do conjunto temos que inicialmente saber a posição deste elemento[1]. Este código é mostrado na figura 12.7.

A união de dois conjuntos consiste em se adicionar todos os elementos dos dois conjuntos e garantir a não repetição de elementos. Isto é meio chato de fazer e é mostrado na figura 12.8.

A intersecção consiste de construir um conjunto que contém apenas os elementos que ocorrem nos dois vetores. O procedimento que faz esta operação é apresentado na figura 12.9.

Finalmente, para sabermos se um conjunto está contido em outro, basta percorrer o primeiro conjunto e verificar se todos os elementos estão no segundo, conforme apresentado na figura 12.10.

Observamos, mais uma vez, que estes algoritmos devem ser alterados e adaptados para o programa do campo minado da seção anterior funcionarem. Isto fica como exercício.

12.2 Tipo Lista

Uma lista pode ser usada em várias situações: lista de compras, lista de pagantes, lista de presentes ou de mebros de um clube. Trata-se de uma coleção de elementos, eventualmente com repetição. Operações importantes sobre listas incluem saber se a

[1]Note que o programa do campo minado precisa de um procedimento que remove um elemento qualquer do conjunto e isto vai ficar como exercício para o estudante.

```
function Cardinalidade (var c: conjunto): integer;
begin
    Cardinalidade:= c.tam;
end;
```

Figura 12.4: Função que retorna a cardinalidade do conjunto.

```
function Pertence (x: real; var c: conjunto): boolean;
var i: integer;
begin
    i:= 1;
    while (i <= c.tam) and (c.v[i] <> x) do
        i:= i + 1;
    if i <= c.tam then Pertence:= true
                  else Pertence:= false;
end;
```

Figura 12.5: Função que define pertinência no conjunto.

lista é vazia, se um elemento pertence ou não a uma lista, inserir ou remover elementos, concatenar listas, e demais operações.

Primeiramente, vamos definir as interfaces, deixando a implementação para um segundo momento. As principais operações sobre listas são as seguintes:

CriaLista (L): Inicializa a estrutura de dados;

ListaVazia (L): Função que rotorna *true* se a lista L é vazia e *false* caso contrário;

Tamanho (L): Função que retorna o número de elementos da lista L.

Pertence (x,L): Função que retorna *true* se o elemento x pertence à lista L e *false* caso contrário;

Insere (x, L): Procedimento que insere o elemento x na lista L;

Remove (x,L): Procedimento que remove o elemento x da lista L;

Uniao (L1, L2, L3): Procedimento que faz a fusão das listas $L1$ e $L2$ retornando o resultado na lista $L3$;

Notem que, contrariamente aos conjuntos, uma lista pode ter repetições, e, por vezes, pode importar a ordem dos elementos.

Podemos implementar uma lista de números reais usando um registro com dois campos: um vetor de reais e o tamanho útil desse.

Neste caso, a implementação das funções e procedimentos recairá nos exercícios que já fizemos no capítulo sobre vetores. Sabemos que as operações de remoção e inserção custam algo proporcional ao tamanho do vetor e que, para piorar, exigem também algo de ordem proporcional ao tamanho do vetor em número de cópias a

```
procedure InsereConjunto (x: real; var c: conjunto);
begin
     if not Pertence (x,c) then
          if Cardinalidade (c) < c.tam then
          begin
               c.tam:= c.tam + 1;
               c.v[c.tam]:= x;
          end
          else
               Erro ('Conjunto cheio');
end;
```

Figura 12.6: Procedimento para inserir elemento no conjunto.

```
procedure RemoveConjunto (x: real; var c: conjunto);
var i: integer;
begin
     (* primeiro acha o elemento no conjunto *)
     i:= 1;
     while (i <= c.tam) and (c.v[i] <> x) do
          i:= i + 1;
     if i <= c.tam then
     begin
          (* se achou remove, senao nao faz nada *)
          for j:= i to c.tam-1 do
               c.v[j]:= c.v[j+1];
          c.tam:= c.tam - 1;
     end;
end;
```

Figura 12.7: Procedimento para remover elemento do conjunto.

cada elemento removido ou inserido no "meio" do vetor, uma vez que os vetores não podem ter "buracos".

Notem que isto ocorre no tipo *conjunto* definido na seção anterior quando se remove um elemento do conjunto, mas não na operação de inserção, que é sempre no final.

Em sua forma básica, um vetor é feito de tal maneira que para cada elemento o seu sucessor é sempre o que está no índice seguinte do vetor. Por exemplo, se um elemento está na posição i do vetor, o próximo estará na posição $i + 1$.

Vamos modificar este comportamento e introduzir uma nova maneira de se colocar as informações no vetor. Agora, o elemento sucessor do que está na posição i poderá ficar em qualquer outra posição diferente de i, bastando para isto que se anote qual é esta posição. Isto exigirá, no momento, o uso de um campo adicional para indicar esta informação, tal como mostrado na figura 12.11.

Para exemplificar, consideremos a lista de elementos como sendo: 5.2, -1.9, 4.8, 3.3, -2.7 e 7.2, nesta ordem. O desenho abaixo mostra essa lista na estrutura proposta.

```
procedure UniaoConjunto (var c1, c2, c3: conjunto);
var i, cont: integer;
begin
     CriaConjunto (c3);
     (* como o primeiro conjunto nao tem repeticao            *)
     (* insere todos os elementos deste no conjunto resultante *)
     for i:= 1 to c1.tam do
          c3.v[i}:=  c1.v[i];
     c3.tam:= c1.tam;
     (* para cada elemento do segundo conjunto, se ele nao    *)
     (* estiver no primeiro, insere no final do segundo       *)
     cont:= 0;
     for i:= 1 to c2.tam do
          if not Pertence (c2.v[i}, c1) then
          begin
               Insere (c2.v[i], c3);
               cont:= cont + 1;
          end;
     c3.tam:= c3.tam + cont;

end;
```

Figura 12.8: Procedimento para unir dois conjuntos.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
			3.3		7.2			5.2		4.8		-1.9					-2.7		
			18		0			13		4		11					6		

Para recuperarmos a lista, basta saber a posição do primeiro, isto é, a posição 9. De fato, o elemento 5.2 está na nona posição do vetor. O elemento seguinte está indicado logo abaixo, isto é, na posição 13. Novamente, conferimos que o segundo elemento da lista, -1.9, está na posição 13 do vetor. O próximo, 4.8, está na posição 4. Pode-se conferir que, de fato, o 3.3 está na quarta posição do vetor e seu sucessor está na posição indicada, 18. O -2.7 está nesta posição e o último, o 7.2 na posição indicada, a sexta. O fato deste elemento ser o último é anotado com o índice zero.

As funções e procedimentos terão que lidar com o tamanho máximo deste vetor, que é definido pela constante *max*. Vamos considerar a existência de um procedimento denominado *Erro (msg)* que retorna erro em caso de qualquer problema apresentando a mensagem *msg*. Evidentemente que isto pode ser feito de outra maneira, mas esta opção é apenas para facilitar a redação do texto.

Vamos implementar na ordem os procedimentos acima descritos. O primeiro deles é o que inicializa a estrutura, que deve resultar na criação de uma lista vazia. Para isto, basta definirmos que o tamanho do vetor é zero, conforme mostrado na figura 12.12.

Assim, é fácil também definirmos o código da função que testa se a lista é ou não vazia, conforme mostrado na figura 12.13.

Também é trivial a função que retorna o número de elementos da lista, o que é apresentado na figura 12.14.

A próxima tarefa é criar um procedimento que define se um elemento faz parte

```
procedure InterseccaoConjunto (var c1, c2, c3: conjunto);
var i, cont: integer;
begin
      CriaConjunto (c3);
      cont:= 0;
      for i:= 1 to c1.tam do
            if Pertence (c1.v[i], c2) then
            begin
                  InsereConjunto (c1.v[i], c3);
                  cont:= cont + 1;
            end;
      c3.tam:= cont;
end;
```

Figura 12.9: Procedimento para fazer a intersecção de dois conjuntos.

```
function ContidoConjunto (var c1, c2: conjunto): boolean;
var i: integer;
begin
      ContidoConjunto:= false;
      i:= 1;
      while (i<= c1.tam) and Pertence(c1.v[i], c2) then
            i:= i + 1;
      if i = c1.tam then
            ContidoConjunto:= true
      else
            ContidoConjunto:= false;
end;
```

Figura 12.10: Procedimento para verificar se um conjunto está contido em outro.

da lista. Para isto, é necessário que se percorra o vetor em busca do elemento. Isto já foi feito nesta disciplina, para o caso de um vetor "tradicional" (ver seção 10.1.3. Mas, desta vez, os elementos estão armazenados de um modo diferente, em primeiro lugar, sem ordenação, o que impede a busca binária. Em sugundo lugar, percorrer a lista é um pouco diferente do que percorrer um vetor, pois o sucessor na lista de um elemento pode estar em qualquer outra posição, logo, o tradicional incremento do apontador deve ser substituído levando-se em conta o campo "próximo". Isto é mostrado na figura 12.15.

A próxima tarefa é garantir a inserção de um elemento na lista. Isto é mostrado na figura 12.16. Uma decisão que precisa ser tomada é onde inserir na lista, isto é, em qual posição. No caso do conjunto, inserimos sempre no final. No caso da lista, o mais fácil é inserir sempre no início, uma vez que temos o apontador para o primeiro elemento. Inserir no final custaria uma busca pelo último elemento, a menos que se modifique a estrutura de dados para se armazenar também o último, além do primeiro.

Assim, optamos por inserir sempre no início, para tornar a operação menos custosa. O estudante é encorajado a construir algoritmos que inserem em uma posição determinada por um parâmetro.

```
const max = 10000;
type
     celula= record
                    elemento: real;
                    proximo: integer;
     end;

     vetor = array [1..max] of celula;

     lista = record
                    tamamho,
                    primeiro: integer;
                    v: vetor;
               end;
```

Figura 12.11: Estrutura de dados para tipo lista.

```
procedure CriaLista (var l: lista);
begin
     l.tamanho:= 0;
end;
```

Figura 12.12: Procedimento para criar uma lista vazia.

Um detalhe adicional é que, nesta estrutura, precisamos saber quais são as posições do vetor que não são ocupadas por nenhum elemento da lista, pois uma inserção vai ocupar um espaço novo. Por exemplo, na ilustração acima, das 20 posições do vetor, apenas 6 estão ocupadas. Vamos supor que existe uma função que retorne um índice não ocupado no vetor:

AlocaPosicao (v): Função que retorna um inteiro entre 1 e max indicando uma posição do vetor que não é usada por nenhum elemento da lista.

Para remover um elemento da lista temos que inicialmente saber a posição deste elemento. Vamos precisar de um procedimento para informar que a posição no vetor do elemento removido da lista agora está livre e pode ser usada futuramente em um processo de inserção.

DesalocaPosicao (p, v): Procedimento que informa que a posição p não está mais em uso e pode ser alocada futuramente.

O problema deste procedimento é que, para remover um elemento da lista, deve-se simplesmente apontar o campo próximo do elemento anterior do que será removido para o sucessor deste. Não há a necessidade de se mover todos os que estão "na frente" para trás, como fazíamos no caso do vetor tradicional.

Localizar a posição do elemento a ser removido não é suficiente, uma vez que temos também que localizar a posição do anterior. Isto exige um código mais elaborado, o qual é mostrado na figura 12.17.

```
function ListaVazia (var l: lista): boolean;
begin
     if l.tamanho = 0 then ListaVazia:= true
                      else ListaVazia:= false;
end;
```

Figura 12.13: Função que testa se lista é vazia.

```
function Tamanho (var l: lista): integer;
begin
     Tamanho:= l.tamanho;
end;
```

Figura 12.14: Função que retorna o número de elementos da lista.

A união de duas listas consiste em se adicionar todos os elementos das duas listas. Basta varrer as duas listas e inserir na terceira, conforme apresentado na figura 12.18.

```
function Pertence (x: real; var l: lista): boolean;
var i: integer;
begin
     i:= l.primeiro; (* indice no vetor do primeiro elemento da lista *)
     while (i <= l.tamanho) and (l.v[i].elemento <> x) do
          i:= l.v[i].proximo; (* indice no vetor do proximo elemento na lista *)
     if i <= l.tamanho then Pertence:= true
                       else Pertence:= false;
end;
```

Figura 12.15: Função que define pertinência na lista.

```
procedure InsereLista (x: real; var l: lista);
var p: integer;
begin
     if Tamanho (l) < l.tamanho then
     begin
          l.tamanho:= l.tamanho + 1;
          p:= AlocaPosicao (v);          (* acha posicao livre *)
          l.v[p].elemento:= x;
          l.v[p].proximo:= l.primeiro; (* o que era primeiro agora eh o segundo *)
          l.primeiro:= p;              (* o novo elemento agora eh o primeiro *)
     end
     else
          Erro ('Lista cheia');
end;
```

Figura 12.16: Procedimento para inserir elemento na lista.

```
procedure RemoveLista (x: real; var l: lista);
var i: integer;
begin
     if ListaVazia (l) then
          Erro ('Lista vazia, nao eh possivel remover elemento')
     else
          if l.v[l.primeiro].proximo = 0 then (* lista tem um unico elemento *)
               if l.[l.primeiro].elemento = x then
               begin
                    l.primeiro:= 0;
                    ..tamanho:= 0;
               end;
          else (* lista tem mais de um elemento, achar a posicao de x *)
          begin
               q:= l.inicio;
               p:= l.v[q].proximo;
               while (p <> 0) and (l.v[p].elemento <> x) do
               begin
                    q:= p;
                    p:= l.v[p].proximo;
               end;
               if l.v[p].elemento = x then (* achou posicao de x *)
               begin
                    l.v[q].proximo:= l.v[p].proximo;
                    l.tamanho:= l.tamanho - 1;
                    DesalocaPosicao (p,v);
               end;
          end;
end;
```

Figura 12.17: Procedimento para remover elemento da lista.

```
procedure UniaoListas (var l1, l2, l3: lista);
var i, cont: integer;
begin
     CriaLista (l3);
     i:= l1.primeiro;
     while i <> 0 do
     begin
          Insere (l1.v[i].elemento), l3);
          i:= l1.v[i].proximo;
     end;
     i:= l2.primeiro;
     while i <> 0 do
     begin
          Insere (l2.v[i].elemento), l3);
          i:= l2.v[i].proximo;
     end;
end;
```

Figura 12.18: Procedimento para unir duas listas.

12.2.1 Exercícios

1. Considere o tipo abstrato de dados lista assim definido:

```
TYPE
    TipoIndice = 1..TAM_MAX_LISTA;
    TipoElemento = RECORD
                        Num : Real;
                        Prox: TipoIndice;
                   END;
    Lista = RECORD
                Tam: integer;
                Primeiro: TipoIndice;
                Vet: ARRAY [1..TAM_MAX_LISTA] of TipoElemento;
            END;
```

   Vamos considerar que os elementos da lista são inseridos em ordem crescente *considerando-se a estrutura de dados*. Por exemplo, em algum momento a lista:

   12, 15, 16, 20, 38, 47

   Pode estar armazenada na estrutura da seguinte maneira:

Tam	6								
Primeiro	4								
Vet	16	20		12	47		15		38	
	2	9		7	0		1		5	

   - Desenhe como estará a estrutura de dados lista após a remoção do elemento "20" da lista seguido da inserção do elemento "11". Note que remoções ou inserções devem manter a lista ordenada *considerando-se a estrutura de dados*.
   - Implemente uma função InsereOrdenado em *Pascal* que receba como parâmetros uma lista do tipo lista e um elemento real e insira este elemento no local correto na lista ordenada, *considerando-se a estrutura de dados*.

2. Preâmbulo: Na aula nós trabalhamos uma estrutura "Lista", que foi assim definida:

```
TYPE Registro = RECORD
                    chave: string[20];
                    (* outros campos *)
                    prox: integer;
                END;
     Lista = ARRAY [1..TAM_MAX_LISTA] of Registro;
```

```
VAR
    L: Lista;
    TamLista: integer; (* Ou seja, a lista nao sabe seu tamanho *)
    Primeiro: integer; (* Diz quem é o primeiro elemento da lista *)
```

Nesta estrutura, nós trabalhamos uma série de procedimentos que manipulavam esta lista. Porém, nestes procedimentos, havia a necessidade de passar como parâmetros, além da lista em sí, o seu tamanho e quem é o primeiro. Isto causa a necessidade de definir as duas variáveis globais, a saber, $TamLista$ e $Primeiro$.

Agora, você deve resolver as questões seguintes:

3. Modifique a estrutura da Lista acima para que seja uma lista de (apenas e tão somente) reais, de maneira que as informações sobre seu tamanho e quem é o primeiro estajam definidas na própria lista, isto é, a única variável global relativa à lista será L.

4. Baseado na sua modificação, faça os seguintes procedimentos e/ou funções (a escolha correta faz parte da prova):

   (a) criar a lista;

   (b) inserir o elemento X na posição pos da lista L;

   (c) retornar o elemento X que está na posição pos da lista L.

5. Com base nos procedimentos e/ou funções acima definidos, faça um programa principal que, usando apenas os procedimentos acima, leia uma seqüência de reais e, em seguida, ordene a lista usando para isto o algoritmos de ordenação por seleção. Observe que, a princípio, neste ítem da prova, você não conhece a estrutura interna da lista, você apenas sabe que é uma lista de reais, e tem acesso aos elementos pelas suas posições.

6. Considere o tipo abstrato de dados lista definido em aula:

```
TYPE
    TipoIndice = 1..TAM_MAX_LISTA;
    TipoElemento = RECORD
                        Num : Real;
                        Prox: TipoIndice;
                   END;
    Lista = RECORD
                Tam: integer;
                Primeiro: TipoIndice;
                Vet: ARRAY [1..TAM_MAX_LISTA] of TipoElemento;
            END;
```

Considerando que os elementos da lista estão ordenados em ordem crescente, implemente uma função RemoveOrdenado em *Pascal* que receba como parâmetros uma lista do tipo lista e um elemento do tipo Real e remova este elemento da lista, mantendo a lista ordenada. Não vale remover e ordenar a lista toda.

Considere as seguintes estruturas de dados:

```
TYPE
     TipoIndice = 1..TAM_MAX_LISTA;

     TipoElemento = Real;

     Lista = RECORD
                  Tam: integer;
                  Vet: ARRAY [1..TAM_MAX_LISTA] of TipoElemento;
             END;
```

Isto define um tipo abstrato de dados chamado "Lista". Listas são comumente usadas em diversos tipos de programas. Trata-se basicamente de uma estrutura qualquer que contém alguns números de interesse, no caso, números reais.

A idéia é esconder detalhes da estrutura e trabalhar apenas com "o que fazer"e não "como fazer". Isto é feito pela definição de um conjunto de funções e procedimentos que manipulam a real estrutura, escondendo detalhes inúteis do usuário (no caso, este usuário é um programador também).

As principais operações que manipulam listas são as seguintes:

- criar uma lista (vazia)
- inserir (no início, no fim, na posição p)
- remover (do início, do fim, da posição p)
- verificar o tamanho da lista
- saber se a lista está vazia/cheia
- imprimeir a lista em ordem
- fundir duas listas
- intercalar duas listas
- pesquisar o elemento da posição p na lista
- copiar uma lista em outra
- particionar uma lista em duas, segundo algum critério

Por exemplo, uma função que cria uma lista na estrutura acima definida poderia ser algo assim:

- Convenciona-se que a lista com tamanho zero está vazia.

- Define-se a seguinte função:

```
   procedure cria_lista (var L: Lista);
   begin
L.tam :=0;
   end; {cria lista}
```

Outro exemplo, para saber se a lista está vazia, pode-se fazer algo parecido com isto:

```
   function lista_vazia (var L: Lista): boolean;
   begin
lista_vazia := (L.tam = 0);
   end; {lista vazia}
```

7. Implemente os outros procedimentos e funções indicados acima, isto é, crie um conjunto de funções e procedimentos que irão constituir uma biblioteca que manipula o tipo abstrato de dados Lista.

8. Implemente um programa que encontra bilhetes premiados do jogo da mega-sena, usando o tipo lista acima definido:

   - Seu programa deverá ter ler de um arquivo texto os bilhetes que concorrem ao premio da mega-sena. Neste arquivo, cada linha tem os números apostados. Note que cada linha pode ter um número diferente de apostas, sendo no mínimo 6 e no máximo 10.
   - Invente uma maneira de indicar ao programa que as apostas terminaram. Considere que a primeira coluna do arquivo contém o identificador da aposta, que é o número do cartão (senão você não sabe quem ganhou).
   - Um outro programa deverá gerar aleatoriamente o arquivo com as apostas.
   - Gere uma lista contendo os números sorteados.
   - Use a estrutura de lista para ler o arquivo com as apostas e use apenas operações com a lista para saber quem ganhou prêmios: quadra, quina, sena, indicando separadamente os identificadores dos premiados. Isto é, imprima uma lista com os ganhadores da mega-sena, da quina e da quadra.
   - Implemente uma função (para fins didáticos!!!) que calcule uma combinação de números "sorteados"que não tenha nenhum ganhador.
   - Implemente uma função que calcule uma combinação de números que maximize o número de ganhadores. Note que as apostas tem entre 6 e 10 números.
   - Implemente uma função (para fins didáticos!!!) que não apenas faça que uma determinada aposta seja a vencedora, mas de maneira que ela minimize o número de outros ganhadores. Note que as apostas tem entre 6 e 10 números.

- É proibido manipular diretamente a lista, imagine que você não faz a menor idéia de que se trata de uma record com um inteiro e um vetor.

9. Em aula nós trabalhamos uma estrutura "Lista", que foi assim definida:

```
TYPE Registro = RECORD
                    chave: string[20];
                    (* outros campos *)
                    prox: integer;
                END;
     Lista = ARRAY [1..TAM_MAX_LISTA] of Registro;

VAR
    L: Lista;
    TamLista: integer; (* Ou seja, a lista nao sabe seu tamanho *)
    Primeiro: integer; (* Diz quem é o primeiro elemento da lista *
```

Nós trabalhamos em aula uma série de procedimentos que manipulavam esta estrutura de lista. Porém, é possível implementar os mesmos procedimentos usando-se uma estrutura de vetor. Como em um vetor o primeiro elemento é sempre o da primeira posição, a variável global $Primeiro$ torna-se desnecessária, bastando apenas mantermos o controle da variável $TamLista$.

Neste exercício, você deve resolver as questões seguintes:

(a) Modifique a estrutura da Lista acima para que seja uma vetor de reais.

(b) Baseado na sua modificação, faça os seguintes procedimentos e/ou funções (a escolha correta faz parte da prova):

i. criar a lista (o que não significa inserir elementos nela);

ii. inserir o elemento X na posição *pos* da lista L;

iii. retornar o elemento X que está na posição *pos* da lista L.

(c) Faça um programa principal que, sem usar informações sobre a estrutura interna da lista a não ser pelo uso de funções e procedimentos definidos por você (se não forem os do exercício anterior, você deve construí-los), faça o seguinte:

i. leia uma seqüência de reais e os insira, um a um, na lista, em ordem numérica crescente;

ii. insira um elemento no final da lista;

iii. retire o primeiro elemento da lista;

iv. ordene a lista pelo método de ordenação por seleção;

v. imprima a lista em ordem numérica decrescente.

10. Defina o tipo abstrato de dados polinomio. A estrutura tem que ser capaz de armazenar o grau e somente os coeficientes não nulos de um polinômio. Por exemplo, você não deve armazenar o monômio de grau 2 no seguinte polinômio:

$$(1 + 2x + 3x^3)$$

Faça um programa em *Pascal* que utilize o tipo abstrato de dados definido, leia dois polinômios p e q, calcule o produto r de p e q, imprima o polinômio resultante, leia um certo número real x, calcule o valor de $r(x)$ e o imprima.

Referências Bibliográficas

[Car82] S. Carvalho. *Introdução à Programação com Pascal*. Editora Campus, 1982.

[Far99] H. e outros Farrer. *PASCAL Estruturado*. Editora Guanabara Dois, 1999. 3a edição Guanabara Dois.

[Gui] Manuais on-line do freepascal. Disponíveis juntamente com o compilador em `http://www.freepascal.org`.

[Knu68] D. E Knuth. *The Art of Computer Programming*, volume 1–3. Addison Wessley, 1968.

[Med05] C. Medina, M.A. Fertig. *Algoritmos e Programação: Teoria e Prática*. Novatec, 2005.

[Sal98] L.M. Salveti, D.D. Barbosa. *Algoritmos*. Makron Books, 1998.

[Tre83] P. Tremblay. *Ciência dos Computadores*. McGraw-Hill, 1983.

[Wir78] N. Wirth. *Programação Sistemática em PASCAL*. Editora Campus, 1978.